NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

CROSS-PLATFORM DEVELOPMENT TECHNIQUES FOR MOBILE DEVICES

by

Arthiemarr M. Mangosing

September 2013

Thesis Advisor:	Thomas Otani
Second Reader:	Loren Peitso

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2013	**3. REPORT TYPE AND DATES COVERED** Master's Thesis
4. TITLE AND SUBTITLE CROSS-PLATFORM DEVELOPMENT TECHNIQUES FOR MOBILE DEVICES		**5. FUNDING NUMBERS**
6. AUTHOR(S) Arthiemarr M. Mangosing		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. government. IRB protocol number ___N/A_____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Business and the military have become increasingly dependent on mobile technology in the last decade. The proliferation of mobile devices provides application developers a new and growing market for providing solutions. Mobile devices run on diverse platforms requiring differing constraints that the developer must adhere to. Thus, extra time and resources must be expended to develop multiple versions of a single application for the different platforms. There have been attempts to minimize the need for these extra costs with mobile cross-platform development environments such as Titanium, PhoneGap, and Corona. They are relatively new to the mobile application building world, and though they have the same goal, their approaches are quite different.

In this thesis, we will provide a detailed analysis of these three cross-platform development tools by using each to develop applications, and then compare each by describing relative strengths and weaknesses.

14. SUBJECT TERMS Mobile development, Cross-platform development, Android development, iOS development			**15. NUMBER OF PAGES** 125
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

CROSS-PLATFORM DEVELOPMENT TECHNIQUES FOR MOBILE DEVICES

Arthiemarr M. Mangosing
Lieutenant, United States Navy
B.S., University of South Florida, 2008

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 2013

Author: Arthiemarr M. Mangosing

Approved by: Thomas Otani
Thesis Advisor

Loren Peitso
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Business and the military have become increasingly dependent on mobile technology in the last decade. The proliferation of mobile devices provides application developers a new and growing market for providing solutions. Mobile devices run on diverse platforms requiring differing constraints that the developer must adhere to. Thus, extra time and resources must be expended to develop multiple versions of a single application for the different platforms. There have been attempts to minimize the need for these extra costs with mobile cross-platform development environments such as Titanium, PhoneGap, and Corona. They are relatively new to the mobile application building world, and though they have the same goal, their approaches are quite different.

In this thesis, we will provide a detailed analysis of these three cross-platform development tools by using each to develop applications, and then compare each by describing relative strengths and weaknesses.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

ADT	Android Developer Tools
API	Application Programming Interface
App	Application
AWT	Abstract Window Toolkit
DoD	Department of Defense
GUI	Graphical User Interface
HTML5	HyperText Markup Language version 5
IDE	Integrated Development Environment
NPS	Naval Postgraduate School
OpenGL	Open Graphics Library
OS	Operating System
REST	Representational State Transfer
SDK	Software Development Kit
UI	User Interface
WYSIWYG	What-You-See-Is-What-You-Get

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I cannot give enough thanks to my thesis advisor, Dr. Thomas Otani, for all the invaluable guidance and constant support that he has provided me the past year.

To soon-to-be Dr. Loren Peitso: Thank you very much for all the valuable feedback. You got me back on track when needed and helped get me to a more-professional final product.

Lastly, thank you to my family, loved ones, and those that matter for always believing in me—even through all the difficult times. I am very blessed to have you all in my life.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Like any other large organization, the United States Department of Defense employs widespread use of mobile devices. In addition to productivity increases in business administration, the effective use of mobile technologies has the potential of dramatically improving warfighting capabilities. For example, mobile technologies can help soldiers identify enemy forces, help engineers order replacement parts more quickly, and help medical corpsman diagnose injuries more reliably while in the combat zone.

The DoD currently has an implementation plan in place that at least "promotes the development and use of mobile *non-tactical* applications within the DoD enterprise" [1]. It understands that:

> …the application of mobile technology into globally integrated operations, the integration of secure and non-secure communications, and the development of portable, cloud-accessing command and control capability will dramatically increase the number of people able to collaborate and share information rapidly. [1]

This includes the establishment of a common mobile application development framework to enable operating system interoperability [1]. The plan to expand the use of mobile devices and applications throughout the DoD makes efficient studies of mobile application development techniques all the more important.

Platforms allow application developers many freedoms to utilizing smartphone and operating system features, and that has made it much easier for developers to make applications that help make tasks easier for everyone and that help everyone interact with each other more efficiently, effectively, and with greater satisfaction.

A. PLATFORM DIFFERENTIATION

There are many challenges to the process of efficient mobile application development. The primary one is the proliferation of mobile platforms. iOS and Android are the two major platforms, but Android has multiple deployed variants and there are many other platforms including Windows, Blackberry, and Symbian. Each of these platforms has their own distinct architecture and programming language that developers must understand and adhere to when creating applications for the specific operating system.

As the mobile market grows and developers write more applications for multiple platforms, the cost for developing code for each platform and then implementing, testing, deploying, and maintaining that code grows in resources and manpower requirements. In order to meet the growing demands for more mobile applications on multiple platforms, we need to find more cost-effective ways to develop these multi-platform applications.

B. CROSS-PLATFORM DEVELOPMENT

A possible answer to cost-effective development and maintenance is *mobile cross-platform tools*. The main purpose of these tools is to allow one codebase to be executed on multiple smartphone platforms, with little (if any) by-platform tailoring necessary.

Mobile cross-platform tools are relatively new to the application-building world. Until the Android OS was released in late 2008, the only relevant smartphone operating system on the market that encouraged third-parties to develop and market mobile applications was Apple's iOS. It was not until Android became the iOS's foremost competition that mobile cross-platform development tools became necessary.

While mobile cross-platform application development tools all have the same goal of helping developers quickly and more-easily create applications for multiple mobile operating systems, their approaches can be quite different.

C. GOALS

We will conduct a comparative analysis on three cross-platform application development tools describing relative strengths and weaknesses. The DoD and other organizations can benefit from this thesis by having a better idea of what kind of tool to consider using when developing their mobile applications for multiple platforms.

This analysis will be based on the construction of common components that applications often have and how that construction compares when we utilize the three chosen tools.

D. SELECTED CROSS-PLATFORM TOOLS

The three mobile cross-platform tools we study in this thesis are Corona Lab's Corona, Appcelerator's Titanium, and Adobe's PhoneGap (with Exadel's Appery.io). Corona is a write-once, build-to-many tool that specializes in easing the effort to develop cross-platform game applications. Titanium allows users to develop native applications across multiple platforms through the translation of JavaScript to the respective platforms’ native code. PhoneGap is a framework for building HTML-based web applications that are wrapped in minimal native code for deployment, and Appery.io is a visual tool that *utilizes* PhoneGap to access native platform features [2, 3]. Chapter III details the reasons we chose these specific tools.

II. BACKGROUND

Two mobile operating systems, the Android OS and Apple's iOS, currently dominate the mobile space. To ensure an understanding of the importance of cross-platform development, this section briefly describes the two platforms, including what constitutes their native look and feel.

We summarize previous research related to streamlining the cross-platform development process. Finally, we describe the cross-platform development lifecycle, which will be used when developing our own applications in the next chapters.

A. MOBILE PLATFORMS

According to statistics from the analyst firm called Vision Mobile, iOS and Android OS mobile platforms have increasingly dominated the smartphone market, taking more than 80 percent of sales today (Figure 1). The stable sales of iPhones and the increasing use of Android-based devices have forced less successful competitors such as Microsoft, Blackberry, and Symbian to at least momentarily cede sales to the strengthening iOS/Android duopoly.

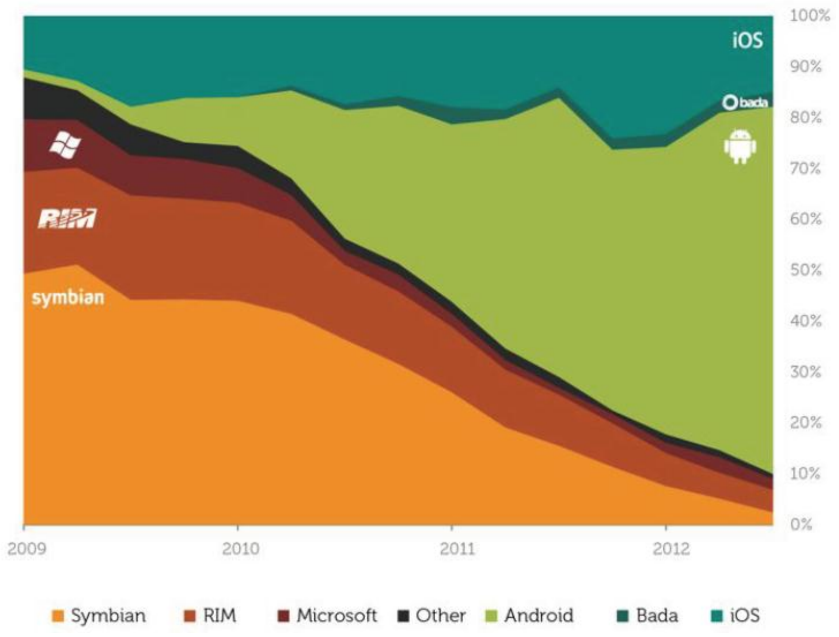


Figure 1. Smartphone Platform Sales. From [4].

1. iOS

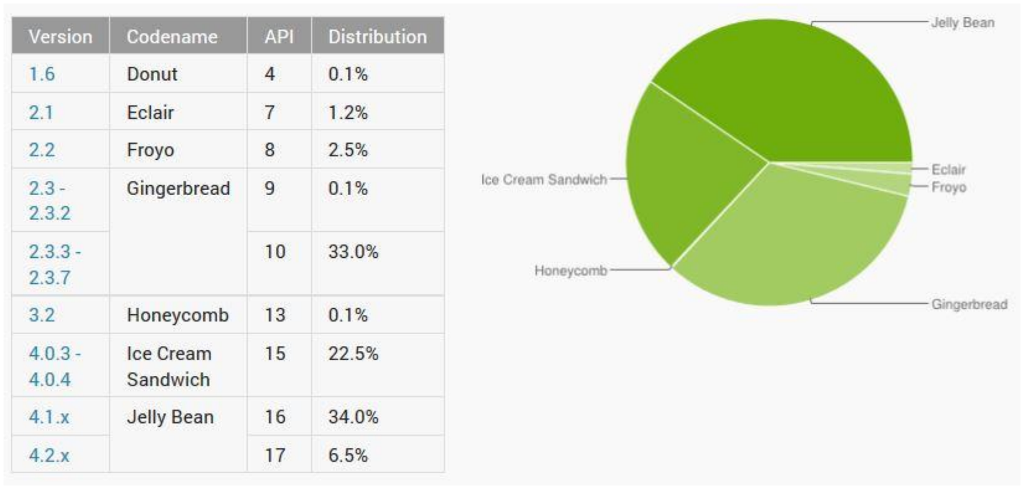
iOS is a mobile operating system used today for the iPhone, iPad, and iPod Touch. Derived from the Mac OS X for desktops, it was released in June 2007 as a focused operating system intended for small display platforms that are also more limited in computer memory resources [5].

Objective-C is the programming language used for iOS application development, and Xcode is the required Integrated Development Environment (IDE). Xcode only runs on Apple Mac OS computers.

2. Android

Google’s Android OS version 1.0 was released in September 2008, after acquiring startup Android, Inc. in 2005 [6]. It continues its frequent updates and improvements with its newest version, Android OS version 4.4 "KitKat", which has not been released as of this writing [7].

While the Android platform has branched out to support different hardware profiles, including screen size, screen resolution, memory size, processor speeds, and pricing options, this fragmentation within the platform also conversely makes it more difficult for developers to support and deploy to all Android devices. Figure 2 details the percentage of Android-based devices running specific Android OS versions.



Version	Codename	API	Distribution
1.6	Donut	4	0.1%
2.1	Eclair	7	1.2%
2.2	Froyo	8	2.5%
2.3 - 2.3.2	Gingerbread	9	0.1%
2.3.3 - 2.3.7		10	33.0%
3.2	Honeycomb	13	0.1%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	22.5%
4.1.x	Jelly Bean	16	34.0%
4.2.x		17	6.5%

Figure 2. Android OS Versions. From [7].

Applications on Android OS are developed "using a subset of Java 6 SE API, replacing Swing, Abstract Window Toolkit, and Applet classes with custom graphics and mobile development libraries" [5]. They can be developed on Windows, Macs, or Linux by using the Android Developers Tools (ADT) plugin with the Eclipse IDE [7]. Applications are made available for download in Google Play.

3. Mobile Application Developer Feedback

Results from a Vision Mobile-conducted survey of 1,200 developers (Figure 3) show that application developers favor iOS over Android OS on five out of seven aspects, including the development environment, documentation and support available, application discovery, potential for revenue, and as the preferred platform to develop on first. The Android OS won in the aspects of having less costs for developers and having a slightly smaller learning curve.

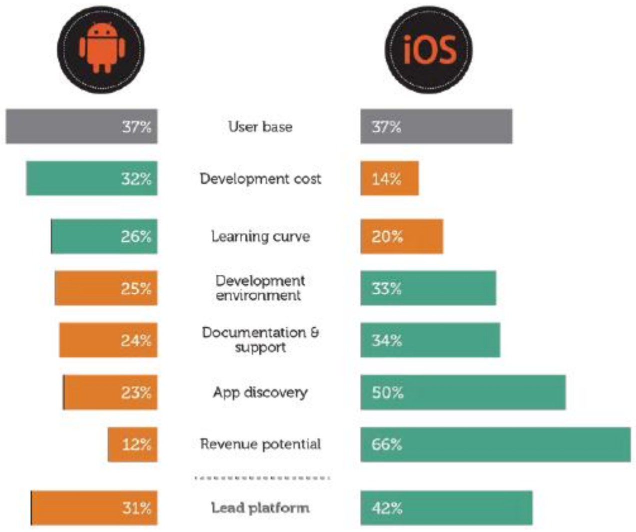


Figure 3. Developers' Platform Preferences. From [4].

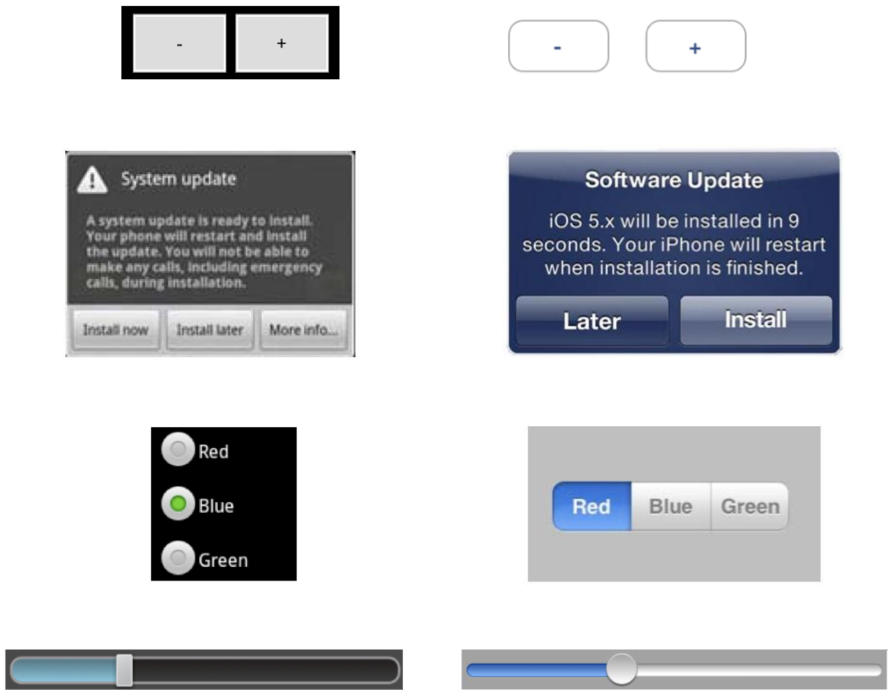
B. PLATFORM LOOK-AND-FEEL

Smartphone users have certain expectations from third-party applications. They expect a distinct platform-consistent user experience that begins when the user opens the platform's home screen. For example, Android users expect the device's back button to always bring them to the previous screen, while iOS users expect a back button to be visible on-screen [8].

The similarly underlying structures for user interfaces (UIs) result in platforms that operate visually much in the same way [5]. Despite this, there are considerable differences in developer-visible aspects and user-visible differences that application developers need to take into consideration to ensure that users' expectations are met. This is a developmental aspect that some cross-platform development tool developers attempt to ease for the application developers.

1. User Interface Differences

Figure 4 highlights some obvious differences between Android OS (left) and iOS 6.x (right) user interfaces. It includes the button widget, alert messages, radio buttons, and slider bars.



A system update is ready to install. Your phone will restart and install the update. You will not be able to make any calls, including emergency calls, during installation.

iOS 5.x will be installed in 9 seconds. Your iPhone will restart when installation is finished.

Figure 4. Platform UI Differences. From [5].

C. PREVIOUS RESEARCH

Previous mobile cross-platform application development research at the Naval Postgraduate School was conducted by Christian G. Acord and Corey C. Murphy. They compared and contrasted the iOS and Android platforms, applied the Model, View, Controller pattern to minimize the differences between the two, and then concocted a unified design process that could be used to implement native iOS and Android applications from a single design process, thereby streamlining the design and implementation for those two platforms together [5]. They focused on a pattern-based approach for development and did not study cross-platform development tools.

D. CROSS-PLATFORM DEVELOPMENT LIFECYCLE

Cross-platform development tools refer specifically to code and graphical UI (GUI) tools that support the simultaneous development of applications for multiple mobile platforms.

There are five stages in the cross-platform application lifecycle [9], and this thesis will compare and analyze the first three.

1. Develop

The develop stage refers to the authoring language (such as JavaScript, Lua, or HTML5), the integrated development environment (IDE), the emulator, and the debugger. The cross-platform tool authoring languages equip developers from backgrounds ranging from traditional software developers to web developers [9].

2. Integrate

The integrate stage refers to integrating the application with platform-specific device capabilities—which can be achieved through the use of tool APIs—as well as connecting to cloud APIs for remote services such as database connectivity [9].

3. Build

There are many technology approaches to completing the build stage, including web-to-native wrappers such as PhoneGap, "application factory" visual design tools like Appery.io, and runtime execution environment application abstracting such as Titanium and Corona [9].

4. Publish

The publishing stage includes submitting the cross-platform application to the target platform application stores. Publishing can also include managing the application store publishing process to some degree [9].

5. Manage

Typically offered by enterprise-targeted cross-platform tools, the application management functionality largely refers to the updating of applications and also includes use of analytics tools. Both Appcelerator and Corona integrate analytics APIs into its tools [9].

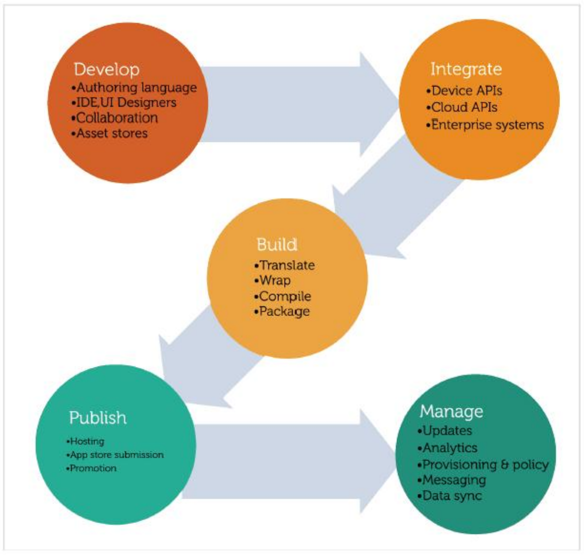


Figure 5. Cross-Platform Application Lifecycle. From [9].

III. APPROACH

To compare and analyze cross-platform application development tools and describe relative strengths and weaknesses, our methodology for this thesis is to first decide which cross-platform tools to study. Next, we develop three typical applications using each of these tools; the application types developed are a productivity application, a game application, and a device-accessing application. During the development process we collect data on the process and tool usage which will be focused on both common application components as well as more general tool observations during the development of these applications.

The smartphone devices used for testing are an iOS-based Apple iPhone 4 and an Android-based Samsung Galaxy S III. For user interface analysis this chapter also includes, from both devices, screen captures of each application that was developed using each tool.

A. CHOSEN DEVELOPMENT TOOLS

Of the over 100 cross-platform development tools in the market today [9], we selected Appcelerator's Titanium, Corona's Corona SDK, and Adobe's PhoneGap (with Exadel's Appery.io) as the tools to be analyzed for the following reasons:

- They fall under differing technology approach categories (described in Chapter 2 under the build stage of the cross-platform application development lifecycle) and can each represent broad categories [9].

- They are well established with funding, company maturity (they are no longer start-up companies), and a large customer base [9].
- Constraints in time that limit the number of tools that we can use for this thesis.

1. Corona

The Corona SDK was created by Corona Labs (previously known as Ansca Mobile) in 2008 to establish itself in the mobile market as a viable contender for iOS and Android cross-platform development [10]. It is a development suite, software development kit (SDK), and runtime environment. Notable characteristics of Corona include 1) its physics and graphics engines, 2) the lightweight Lua scripting language, 3) building applications on Corona Labs' servers rather than locally, 4) a one-click emulator that allows for quick testing, and 5) detailed and well-organized documentation and third-party support.

While its focus is largely on cross-platform game-oriented applications, Corona has expanded its support for other types of applications, including business applications and the ability to access platform features [10].

a. Architecture/Process

Though Corona is a closed-source engine and development kit, leaving developers access to little more than the interface, we were able to learn a bit about the architecture and process. When building code, the Lua script is optimized and pre-compiled into bytecode before being sent to the server. The server then embeds the bytecode into the Corona C/C++ based engine, which includes a rendering engine, OpenGL (Open Graphics Library), OpenAL (Open Audio Library), and Box2d (for physics) [11]. Figure 6 visualizes the Corona Engine and the layers above it. The rest of the process is not available to the public.

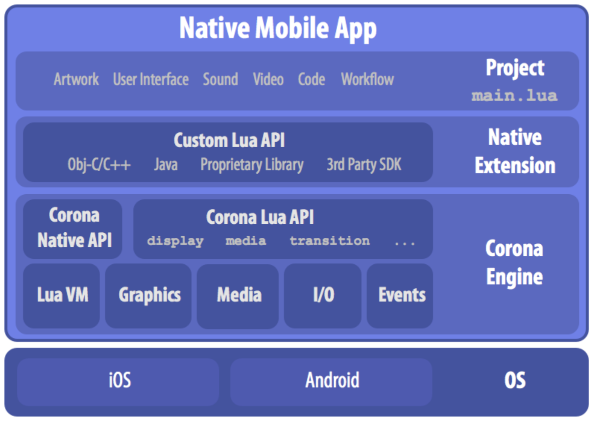


Figure 6. Corona Architecture. From [11].

b. Development Options

There are three subscription options for developers who wish to use Corona SDK: Starter, Pro, and Enterprise. Starter is used for this thesis, which is the free option and allows us the use of the emulator and debugger, the testing of builds on actual devices, access to many developer resources (which connected us to others in the Corona community), and the publishing of Corona applications to application stores as well. The Pro version costs $49 monthly and adds premium support, in-app purchases, and daily software updates on top of what Starter offers. The most-expensive Enterprise version costs $199 monthly but adds features such as allowing developers to access native libraries and to create native-based Corona APIs [10]. Though premium support and access to native libraries may have provided some benefit to our analysis, the Starter option provides us with enough data points to effectively compare Corona SDK to our other selected tools.

2. Titanium

Appcelerator's Titanium SDK began supporting mobile cross-platform development in 2009. Like Corona, it has a one-codebase-to-multiple-platforms philosophy. One of its many differences is that it provides access to numerous native application program interfaces (APIs), thus making many user interface (UI) interactions, UI animations, UI effects, and some platform specific calculations completely native [12].

JavaScript is Titanium's primary development language (though HTML and CSS are also options). The Titanium development environment is an Eclipse-based IDE.

Titanium has access to most of the iOS and Android platform features. Support for Blackberry and Windows Phone are planned [12] in this tool's attempt to reuse code with a unified JavaScript API while at the same time supporting unique features of the specific devices.

a. Architecture/Process

The Titanium architecture includes 1) the JavaScript source code inlined into a Java or Objective-C file and compiled as an encoded string, 2) the implementation of the Titanium API in the platform's native programming language, and 3) a JavaScript interpreter.

When launched, an application has a JavaScript execution environment created in native code. Inside that environment are JavaScript objects which all have a paired object in the native code. These objects are what act as the bridge between JavaScript and the native code [2]. Figure 7 visualizes the architecture of Titanium SDK.

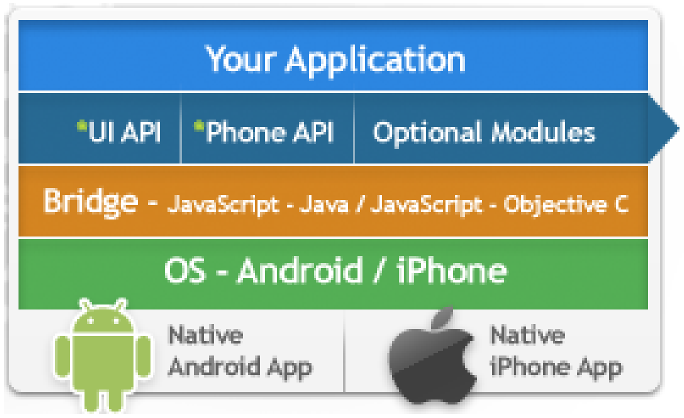


Figure 7. Titanium Architecture. From [2].

b. Development Options

Titanium is open source with a permissive license, so developers have freedoms to modify and extend the framework with functionalities they need. It can be downloaded free but there are many different pricing options for additional features like security and other enterprise extensions. Of note, popular applications that were created from Appcelerator's Titanium include NBC's application and official applications for various universities [12].

3. PhoneGap

Adobe's PhoneGap began development in 2008 at an iPhoneDevCamp [13]. This product creates a web application embedded in a native wrapper, where it is then treated as a native application. The native wrapper provides JavaScript APIs to give access to the respective platform's features, such as contacts, the file system, camera, microphone, and GPS. The application is built using HTML, CSS, and JavaScript, logically targeting those with web development experience. Phonegap supports iOS, Android, Blackberry, Windows, Symbian, and Bada [13].

This thesis will use the PhoneGap-implementing cross-platform tool called Appery.io for PhoneGap-related research.

a. Architecture/Process

A PhoneGap application is a web application "wrapped" in native code. This means that JavaScript files are included in a page loaded by the native device's web browser widget (also known as a web view). A JavaScript API is created inside a web view which can asynchronously send and receive messages with the native wrapper code. This "bridge" is implemented differently in each platform, and it is what allows local web applications to call native code and the native device's features [2]. To ensure clarity, note that there are two instances where JavaScript is used: the business logic part, which drives the UI and its functionality, and the part which accesses and controls the given device.

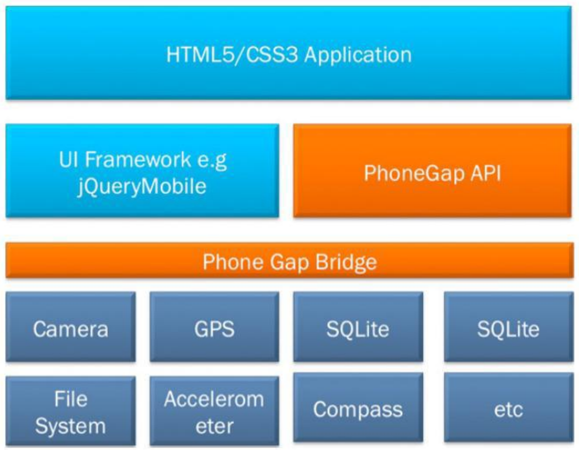


Figure 8. PhoneGap Architecture. From [14].

b. *Development Options*

PhoneGap, like Titanium, is open source with a permissive license. There are tool options that utilize and build on top of PhoneGap such as PhoneGap Build and Appery.io. PhoneGap Build is a cloud-based building option built on top of PhoneGap, which is what we employ for our PhoneGap game application.

Products to note that have been made with the help of PhoneGap include the BBC Olympics application and Wikipedia [13].

c. *Appery.io*

Exadel's Appery.io (previously known as Tiggzi) is a visual, what-you-see-is-what-you-get (WYSIWYG), cloud-based development environment that employs PhoneGap as the bridge to access device-native features [3]. It requires no downloads to use, and uniquely takes advantage of QR codes to give developers the ability to quickly test their products on their phones.

B. METRICS

An important aspect of developer productivity is how well tools will let the developer specify elements common in most applications. Three common components that many (if not most) applications have are the user interface, program control, and data management. Thus, this thesis will examine these tools and the differences between how each tool can construct each component. Other metrics that are related to these three include the amount of setup a developer must invest in, the number of lines of code that components consist of in each tool, and the level of abstraction offered.

1. User Interface

The component that is visible to the application user is the user interface. This includes widgets such as those detailed in Chapter 2.

2. Program Control

Program control is the component that refers to domain logic, control logic, and event handling. It is the part that responds to the end-user’s commands and also performs automated tasks that are structured into the application [15].

3. Data Management

Data management refers to the method that an application uses to access and query databases. This includes both onboard and off-board memory storage.

C. DEVELOPED APPLICATIONS

We describe the three applications that we developed using each of the three tools. These applications were chosen because they apply the kind of aspects that we expect in a mobile application. For each application, we will describe the reasoning behind developing it, the desired design, and the application from the perspective of the user.

1. Productivity Application

Productivity applications represent mobile software that have the goal of enhancing time efficiency and task success. They include tools that help users create and manage documents, organize schedules, store data via cloud services, calculate finances, and increase communication options.

The productivity application that we have chosen to create is a recipe collection application. It gives users a simple method of organizing recipes by use of database operations via an intuitive UI. Users can view recipes by name in a list and then select a recipe for a more detailed explanation. Users can also add new entries and delete old entries from the application.

This application allows us to study how the tools support data-oriented operations. It allows us to study the key APIs (including lists, forms, buttons, scene changing, and database operations) that may be notably different in each tool. It also gives us many opportunities to test the differences between each of tools in regards to how they handle each of our component metrics.

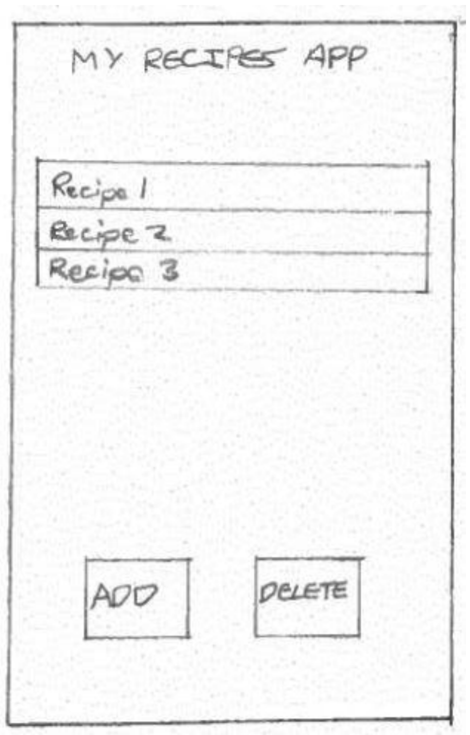


Figure 9. Productivity Application List.

Figure 9 shows a list of recipes stored in the database. When a user selects an item, the application will locate the required data (the recipe instructions) and display it in a customized display panel (Figure 10, left). The user can also delete an entry by tapping the delete button along with the item to delete. Users can add a new recipe by tapping the add button whereupon the screen will transition to a different view for inputting a recipe’s name and instructions (Figure 10, right).

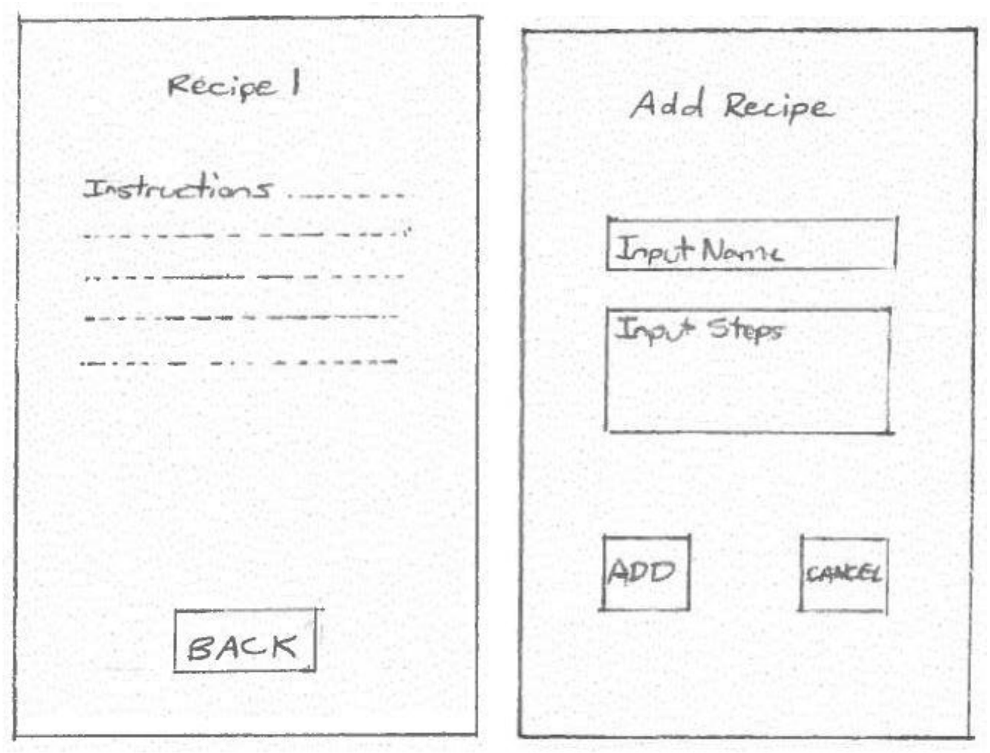


Figure 10. Productivity Application Instructions and Add.

The instructions screen and the add screen both have a back/cancel button that transitions the user back to the list page. The add screen has two textboxes for user input as well as an add button for the user to submit the recipe to the database and then go back to the list screen where they can view all the recipes—including the latest addition.

2. Game Application

Mobile game applications dominate the mobile world. Development of a game application was chosen not just for its market size, but its relevance to the Department of Defense; the DoD relies heavily on simulations, which share many aspects of game applications. Techniques necessary to build games are also directly applicable to building many of the DoD relevant applications.

With our game application, we highlight event-handling as well as the physics and graphics components of each of the three cross-platform development tools, including their levels of abstraction.

The game that we have chosen to create is a simple “ball catching” application. It begins with a ball falling from the top of the screen and a timer counting down until the game is over. The goal of the game is to repeatedly tap the ball in order to keep it from falling out of view. Tapping the ball causes it to travel to a random new position on the screen where it will continue its fall.

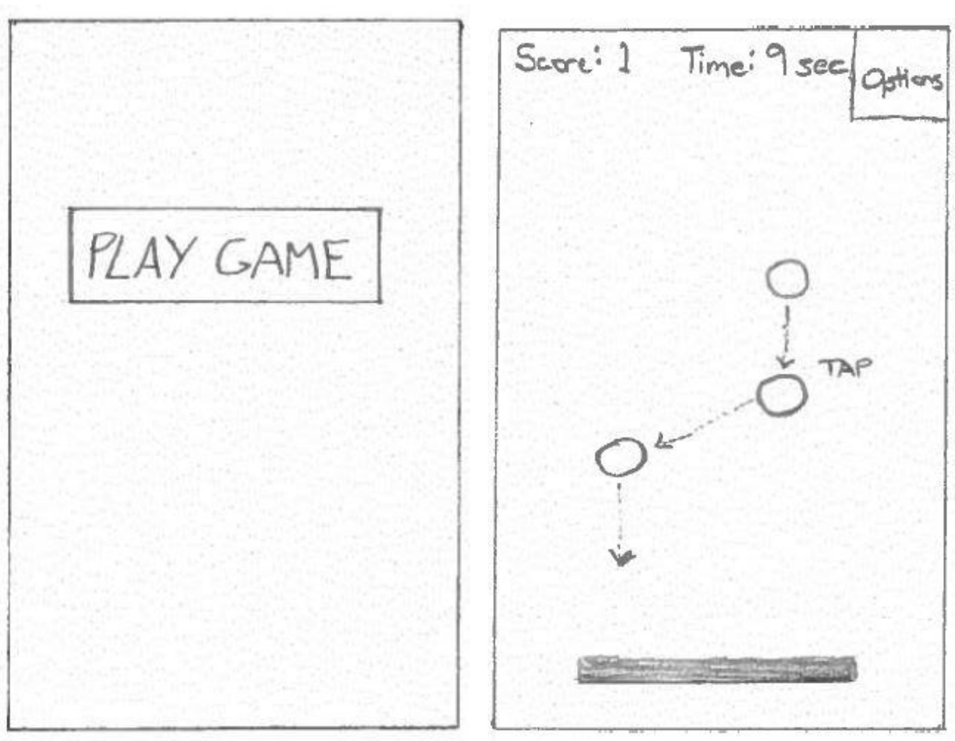


Figure 11.   Game Application.

Figure 11 (left) is a start screen that introduces the game. The user only needs to tap the screen to begin playing, the first of multiple event-handlers.

Figure 11 (right) is the game screen. The top of the screen includes the score, the countdown timer, and an options button for the user to access the settings screen. There is a platform near the bottom on which the ball will bounce upon collision, helping the player to keep the ball on the screen. After the timer has ended, the user's final score is displayed.

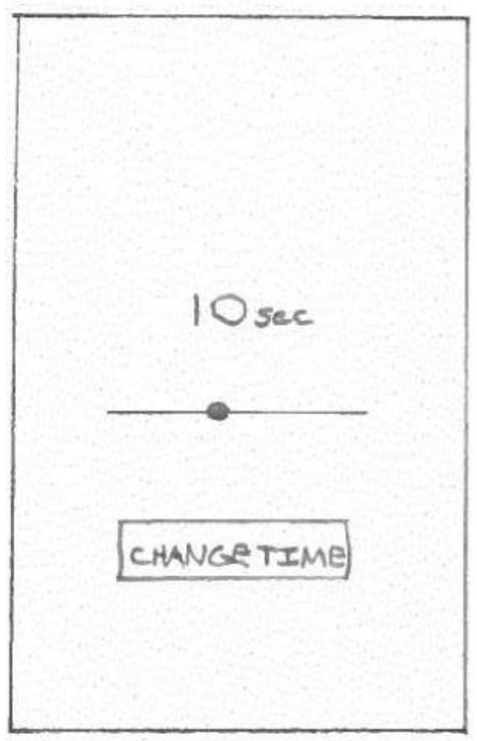


Figure 12. Game Application Options.

The options screen (Figure 12) includes a slider bar so that the user can change the number of seconds in a game, representing data handling from screen to screen. Tapping the confirm button applies the change.

3. Device-Accessing Application

Development of a device-accessing application was chosen because, for a mobile development tool to be truly cross-platform, we have to be able to take advantage of a mobile device's physical features.

The application that we will develop utilizes each cross-platform development tool's ability to apply smartphones' geolocation and camera features. It pinpoints the devices location as well as the location of any input addresses onto a map. Both features are supported by key APIs and, thus, they are aspects of both the user interface and program control components.

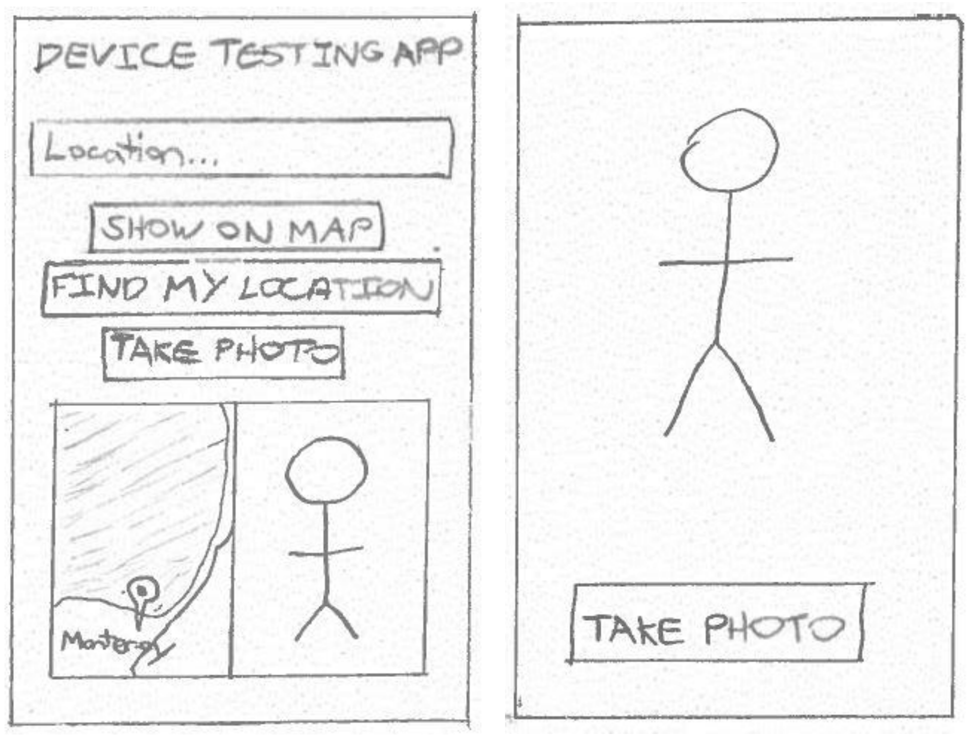


Figure 13.   Device-Accessing Application.

Figure 13 is the design concept. Under the name of the application is a textbox where the user can enter a location. Upon selecting the "show on map" button, the application will display the given location on the onscreen map. When the user selects the "find my location" button, the application utilizes the device's geolocation feature to determine its location and pinpoint it on the same map.

If the user selects the "take photo" button, the application will go to the camera screen where the user can take a photo by utilizing the device's camera feature.

D. HYPOTHESIS AND EXPECTATIONS

When the three tools are compared to each other based on each metric we expect the following:

- In regards to the user interface component, each of our tools (based on our device screen captures) has already proven that they offer the functionalities necessary for developers to create similar UIs to their counterparts. The level of program abstraction offered by tools will help determine the ease that developers will have in developing the UI as well as the other components.
- For the program control component, event-handling is where we gain the most experience for our analysis. While the Corona tool—due to its focus and relative maturity—will offer the most APIs for physics and graphics, Titanium will give developers the most ease to handling native widget-related event functions.
- In regards to the data management component, the programming language and method used will affect the results. Though Appery.io (with PhoneGap) will provide the most ease for database use, Titanium and Corona with their use of SQLite3 will still be able to handle the same operations.

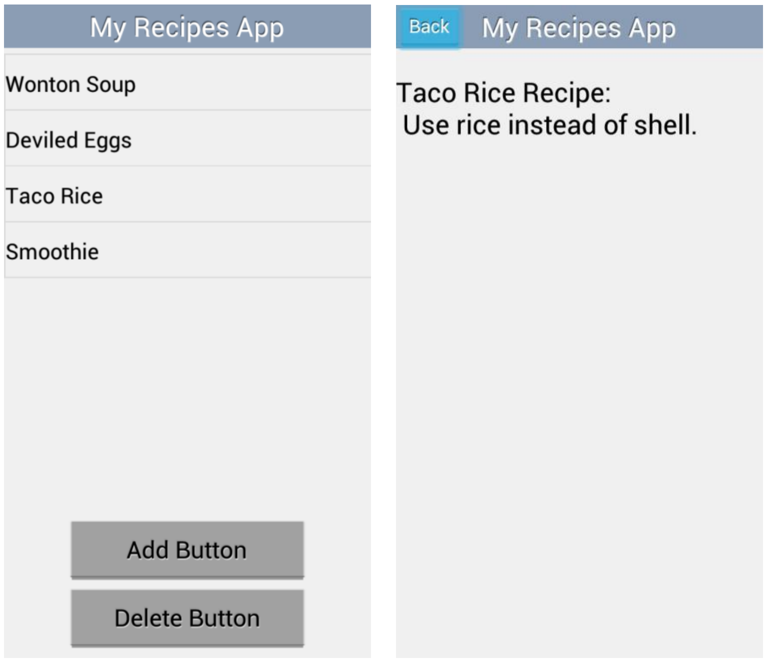
IV. APPLICATION DEVELOPMENT PROCESS

In this chapter we detail each of our iOS/Android cross-platform applications that we developed with each tool, describing especially important portions of our application code and elaborating on code worth analyzing.

A. CORONA

1. Productivity Application

Figures 14 and 15 are screenshots of the final productivity application as seen on the Samsung Galaxy S3 smartphone. Figures 16 and 17 show the same scenes but the screenshots were taken from Apple’s iPhone 4. The only difference in the code used in each platform was the theme set, which are Lua files with theme tables that correspond to individual widgets [16]. This is Corona’s attempt to provide a native look to application UIs when running on either iOS or Android platforms. Note how the slider objects are tailored to match the operating system. These are loaded images made to imitate the look of sliders that are created natively.



Taco Rice Recipe:
Use rice instead of shell.

Figure 14. Corona Android Productivity Application.

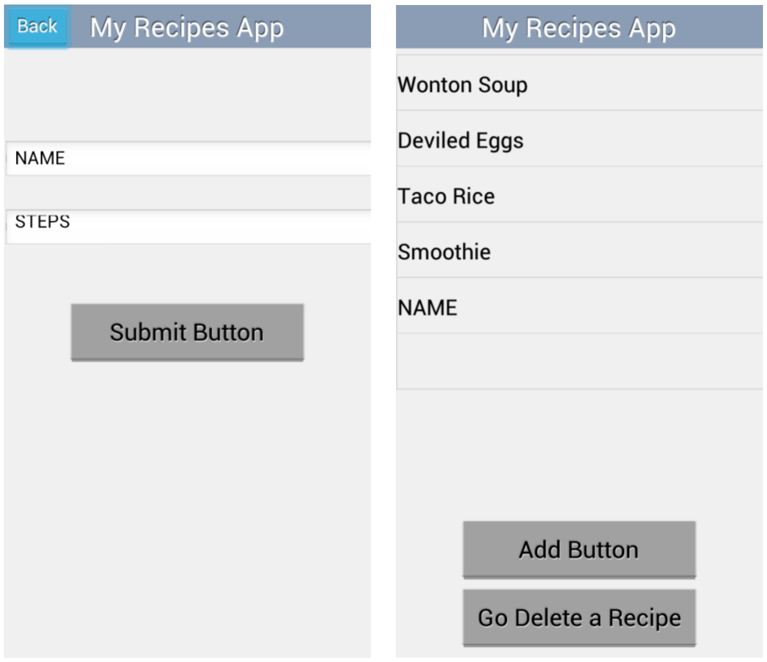


Figure 15. Corona Android Productivity Application cont.

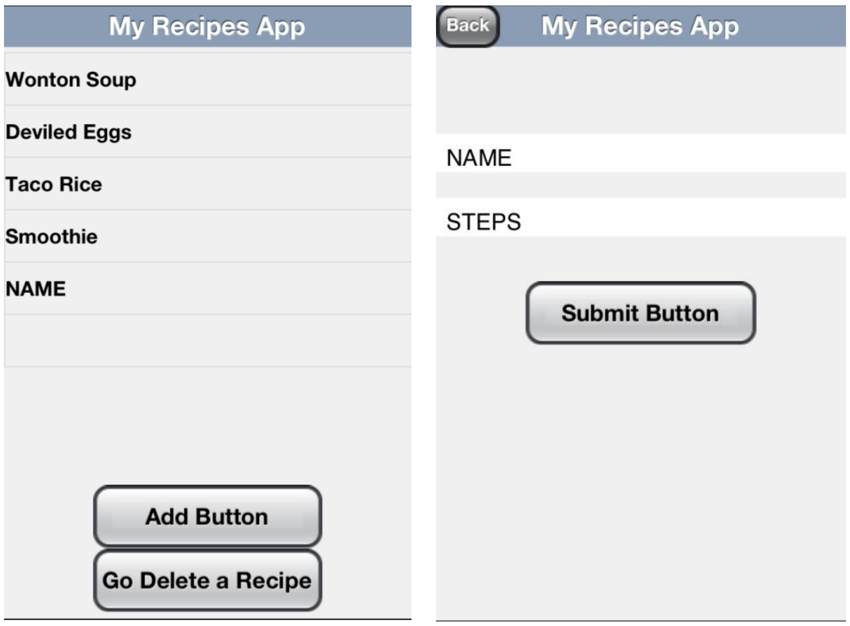
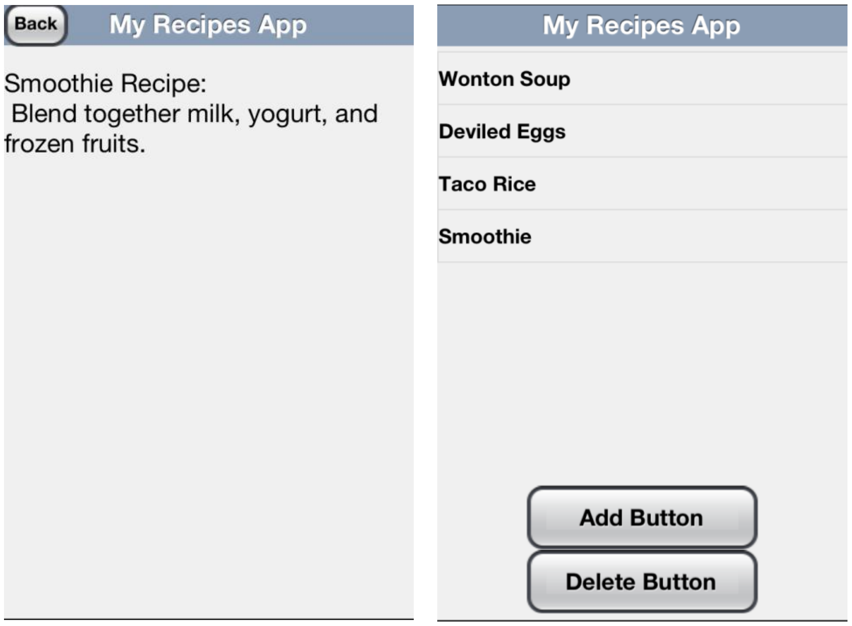


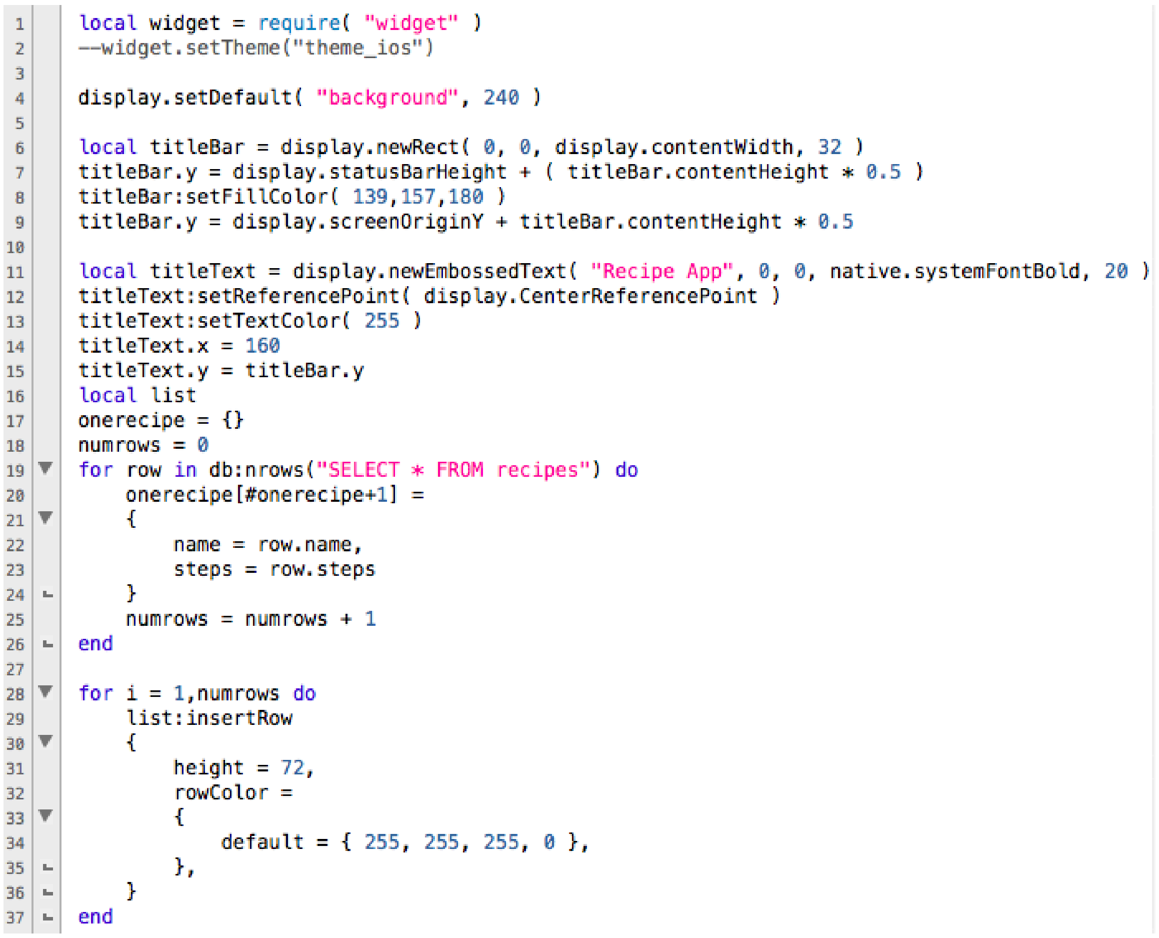
Figure 16. Corona iOS Productivity Application.



Smoothie Recipe:
 Blend together milk, yogurt, and frozen fruits.

Figure 17. Corona iOS Productivity Application cont.

In the main window, there is a table view and two buttons. Figures 18 and 19 include the Lua statements to construct the main window.



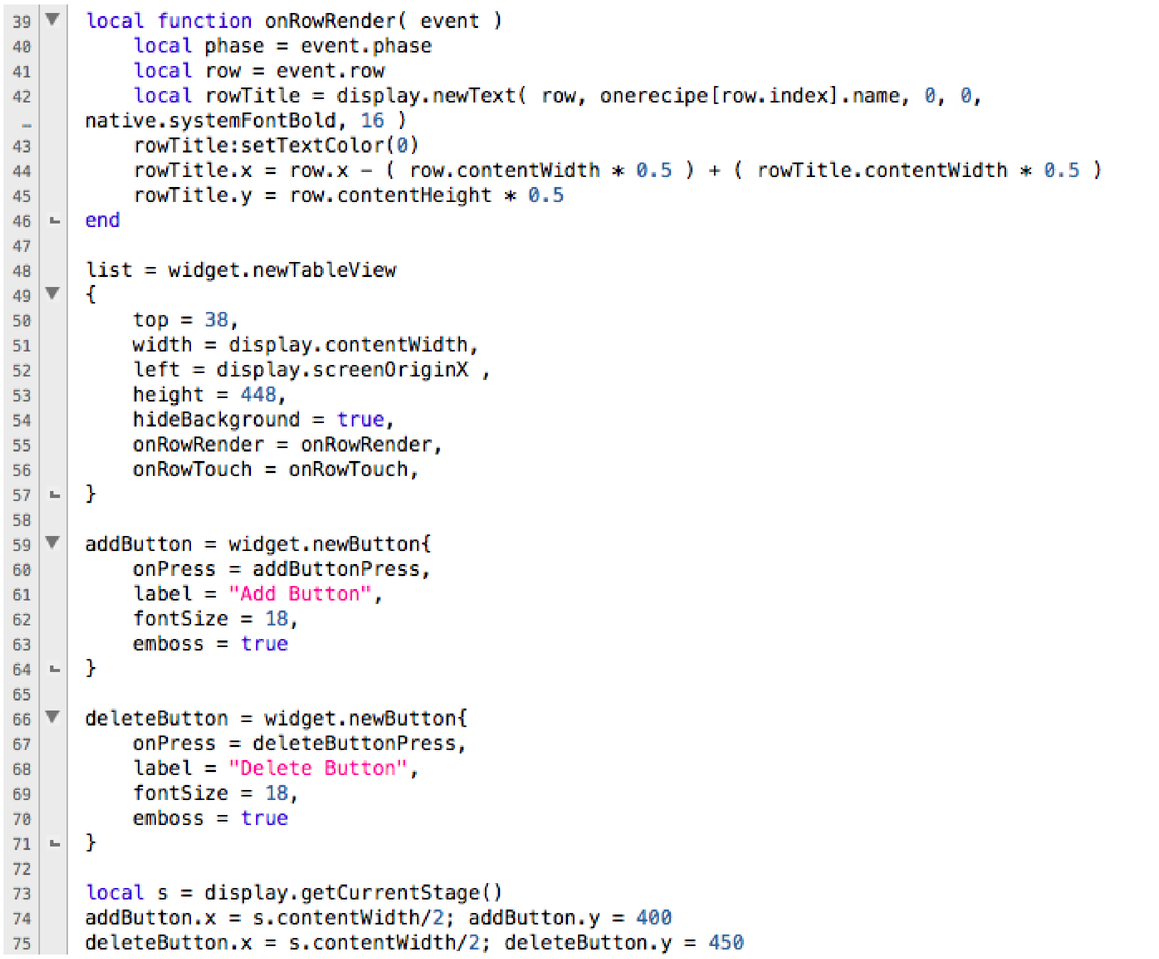
```lua
local widget = require( "widget" )
--widget.setTheme("theme_ios")

display.setDefault( "background", 240 )

local titleBar = display.newRect( 0, 0, display.contentWidth, 32 )
titleBar.y = display.statusBarHeight + ( titleBar.contentHeight * 0.5 )
titleBar:setFillColor( 139,157,180 )
titleBar.y = display.screenOriginY + titleBar.contentHeight * 0.5

local titleText = display.newEmbossedText( "Recipe App", 0, 0, native.systemFontBold, 20 )
titleText:setReferencePoint( display.CenterReferencePoint )
titleText:setTextColor( 255 )
titleText.x = 160
titleText.y = titleBar.y
local list
onerecipe = {}
numrows = 0
for row in db:nrows("SELECT * FROM recipes") do
    onerecipe[#onerecipe+1] =
    {
        name = row.name,
        steps = row.steps
    }
    numrows = numrows + 1
end

for i = 1,numrows do
    list:insertRow
    {
        height = 72,
        rowColor =
        {
            default = { 255, 255, 255, 0 },
        },
    }
end
```

Figure 18. Corona Productivity Application Code—Main Window.

Here the widget library is included and platform themes are set. The widget library includes (but is not limited to) buttons, textboxes, and table views (lists), all of which we are using in this particular application.

A titlebar is created to go at the top of the screen as well as the embossed text to go on the titlebar. The titlebar is simply a colored rectangle object while the text is a text object positioned directly over the rectangle.

Figures 18 through 20 show the code to display objects and widgets on the main window.



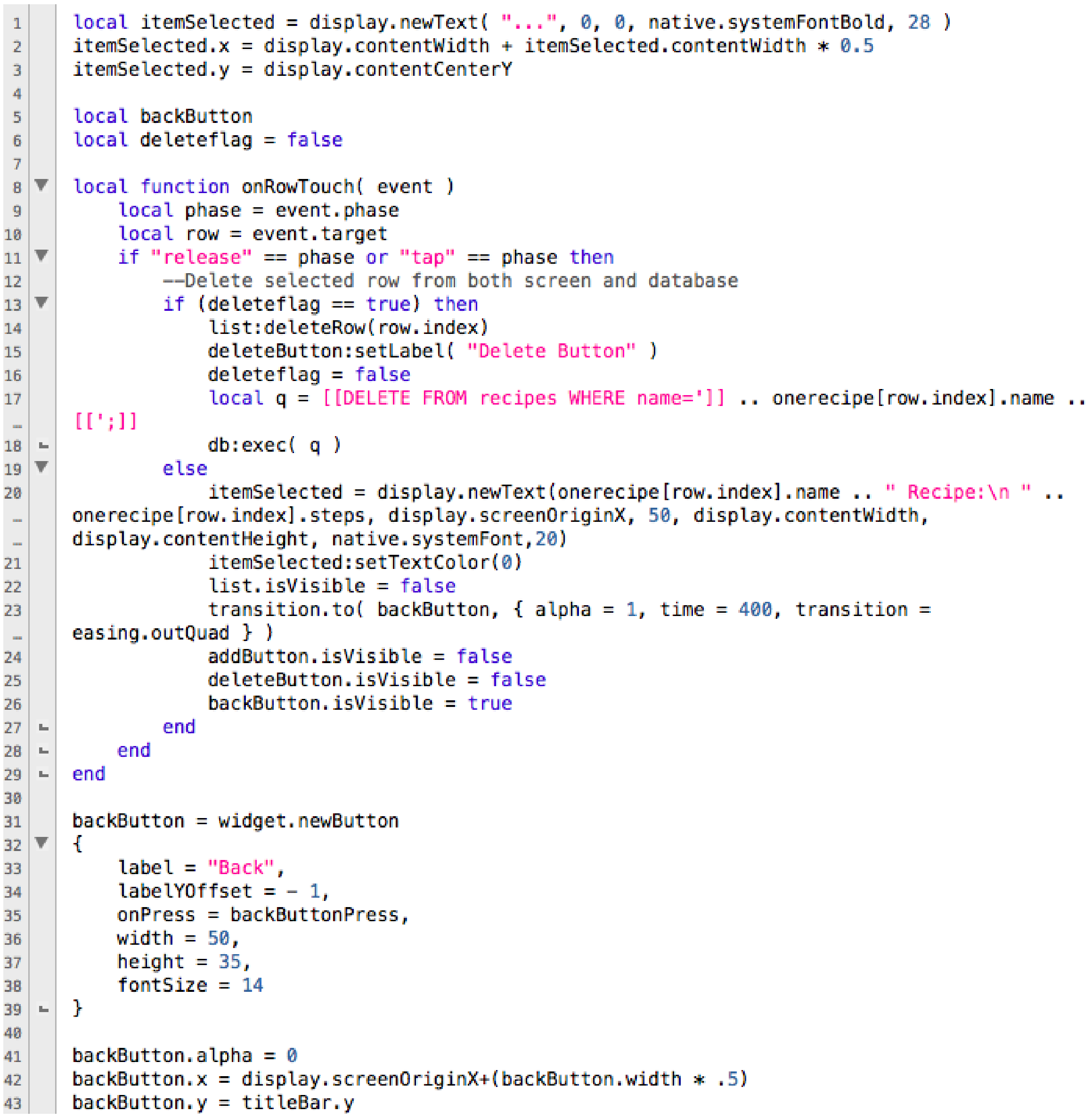
```
local function onRowRender( event )
    local phase = event.phase
    local row = event.row
    local rowTitle = display.newText( row, onerecipe[row.index].name, 0, 0,
native.systemFontBold, 16 )
    rowTitle:setTextColor(0)
    rowTitle.x = row.x - ( row.contentWidth * 0.5 ) + ( rowTitle.contentWidth * 0.5 )
    rowTitle.y = row.contentHeight * 0.5
end

list = widget.newTableView
{
    top = 38,
    width = display.contentWidth,
    left = display.screenOriginX ,
    height = 448,
    hideBackground = true,
    onRowRender = onRowRender,
    onRowTouch = onRowTouch,
}

addButton = widget.newButton{
    onPress = addButtonPress,
    label = "Add Button",
    fontSize = 18,
    emboss = true
}

deleteButton = widget.newButton{
    onPress = deleteButtonPress,
    label = "Delete Button",
    fontSize = 18,
    emboss = true
}

local s = display.getCurrentStage()
addButton.x = s.contentWidth/2; addButton.y = 400
deleteButton.x = s.contentWidth/2; deleteButton.y = 450
```

Figure 19. Corona Productivity App Code—Main Window 2.

The onRowRender() function is a listener for the tableview where, when the list's insertRow function is called, it renders a new row. The list's insertRow function is called for every index in onerecipe. The Add and Delete button widgets are created and positioned on screen.

Events in Lua, which are the main way of creating interactive applications in Corona, are handled similarly to how they are in Titanium's JavaScript: by placing an event as an argument in functions such as **local function name(event).**

When an item in the list is selected (by the end-user tapping on it), the second window appears on the screen to show the details of the selected item. This transition from one window to another is achieved by handling the onRowTouch() event. onRowTouch() relies on a flag's condition as to whether it will delete a touched row (including the associated table row and array index) or switch the screen to the recipe steps screen. The relevant code for this transition is as follows (Figure 20):



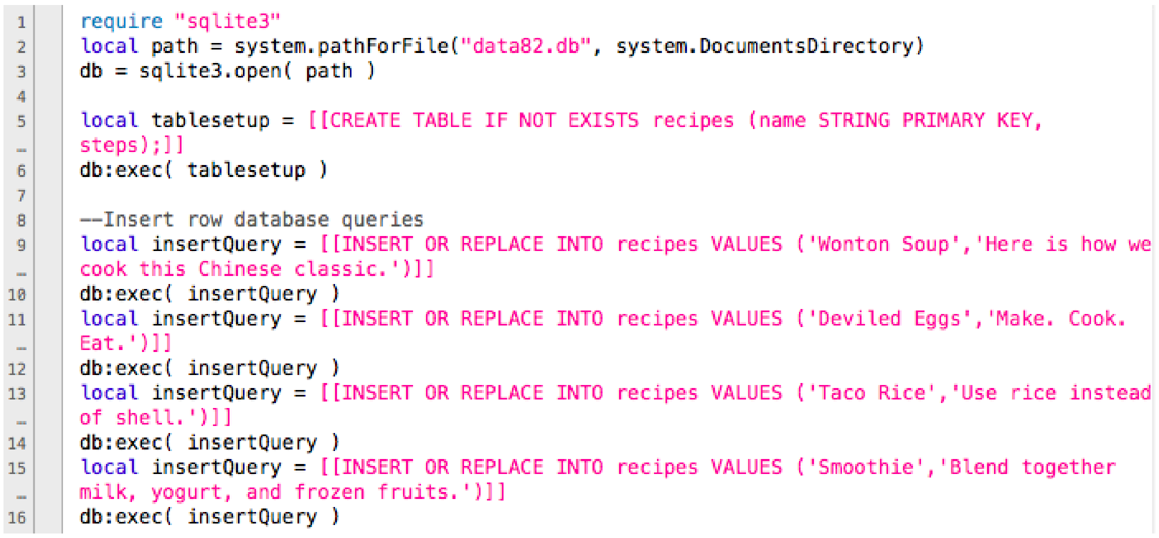
```
local itemSelected = display.newText( "...", 0, 0, native.systemFontBold, 28 )
itemSelected.x = display.contentWidth + itemSelected.contentWidth * 0.5
itemSelected.y = display.contentCenterY

local backButton
local deleteflag = false

local function onRowTouch( event )
    local phase = event.phase
    local row = event.target
    if "release" == phase or "tap" == phase then
        --Delete selected row from both screen and database
        if (deleteflag == true) then
            list:deleteRow(row.index)
            deleteButton:setLabel( "Delete Button" )
            deleteflag = false
            local q = [[DELETE FROM recipes WHERE name=']] .. onerecipe[row.index].name .. [[';]]
            db:exec( q )
        else
            itemSelected = display.newText(onerecipe[row.index].name .. " Recipe:\n " .. onerecipe[row.index].steps, display.screenOriginX, 50, display.contentWidth, display.contentHeight, native.systemFont,20)
            itemSelected:setTextColor(0)
            list.isVisible = false
            transition.to( backButton, { alpha = 1, time = 400, transition = easing.outQuad } )
            addButton.isVisible = false
            deleteButton.isVisible = false
            backButton.isVisible = true
        end
    end
end

backButton = widget.newButton
{
    label = "Back",
    labelYOffset = - 1,
    onPress = backButtonPress,
    width = 50,
    height = 35,
    fontSize = 14
}

backButton.alpha = 0
backButton.x = display.screenOriginX+(backButton.width * .5)
backButton.y = titleBar.y
```

Figure 20. Corona Productivity App Code—Scene Transition.

The recipe data are managed by the backend relational database called sqlite3, which is a lightweight database management system optimized for limited-capacity devices. The recipe database is created when the application is executed for the first time. We detect the existence of a database at the program startup. If no database is detected, it is created. If the database already exists, then it is used in the program.

Back in Figure 18 we go through the database table via the SELECT* query and store each entry into a Lua list, also keeping track of the number of recipes so that we can properly display them in the UI tableview (list) [17].

The database detection and creation is done as follows (Figure 21)[18]:

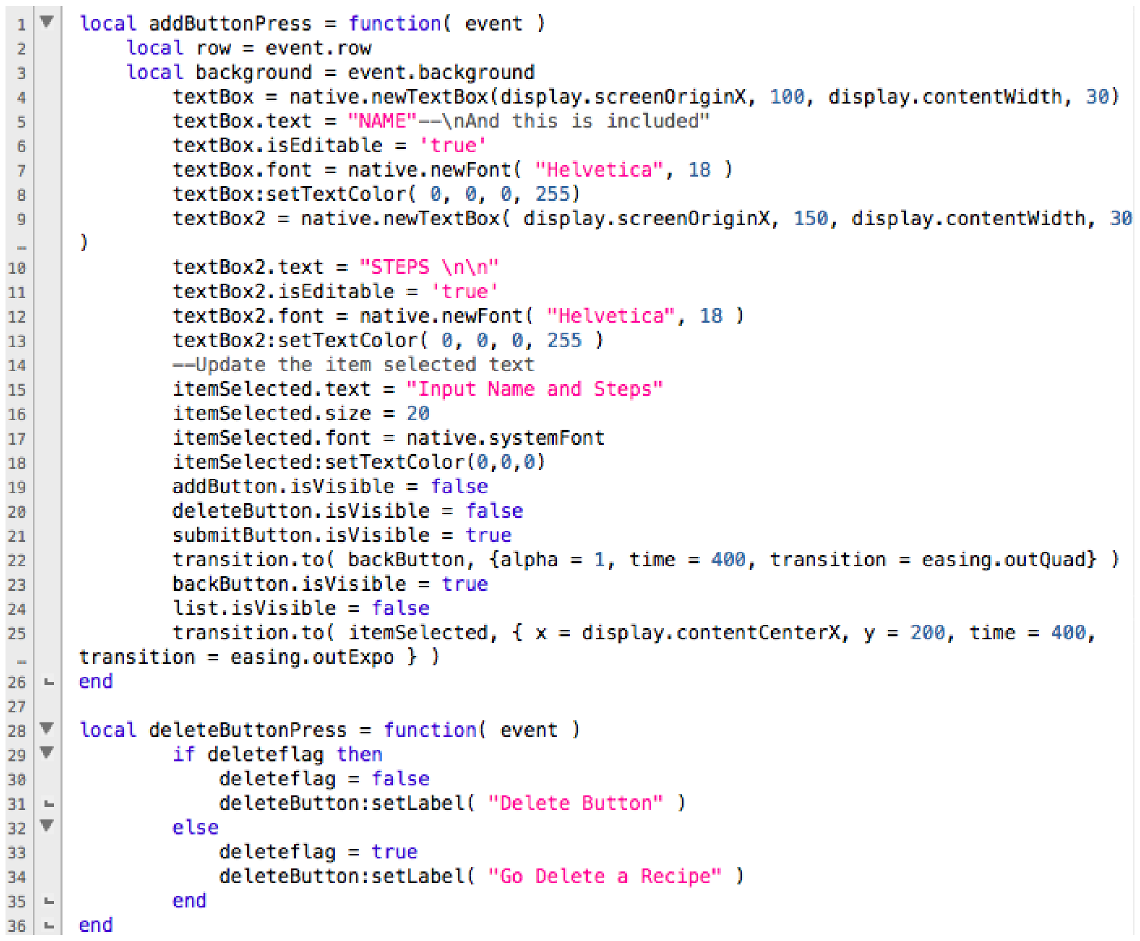


```
require "sqlite3"
local path = system.pathForFile("data82.db", system.DocumentsDirectory)
db = sqlite3.open( path )

local tablesetup = [[CREATE TABLE IF NOT EXISTS recipes (name STRING PRIMARY KEY, steps);]]
db:exec( tablesetup )

--Insert row database queries
local insertQuery = [[INSERT OR REPLACE INTO recipes VALUES ('Wonton Soup','Here is how we cook this Chinese classic.')]]
db:exec( insertQuery )
local insertQuery = [[INSERT OR REPLACE INTO recipes VALUES ('Deviled Eggs','Make. Cook. Eat.')]]
db:exec( insertQuery )
local insertQuery = [[INSERT OR REPLACE INTO recipes VALUES ('Taco Rice','Use rice instead of shell.')]]
db:exec( insertQuery )
local insertQuery = [[INSERT OR REPLACE INTO recipes VALUES ('Smoothie','Blend together milk, yogurt, and frozen fruits.')]]
db:exec( insertQuery )
```

Figure 21. Corona Productivity Application Code—Database.

A variable that holds the "create database table" query is executed. Then, variables that hold "insert database row" queries are executed. The database table includes both the recipe names and the recipe steps.

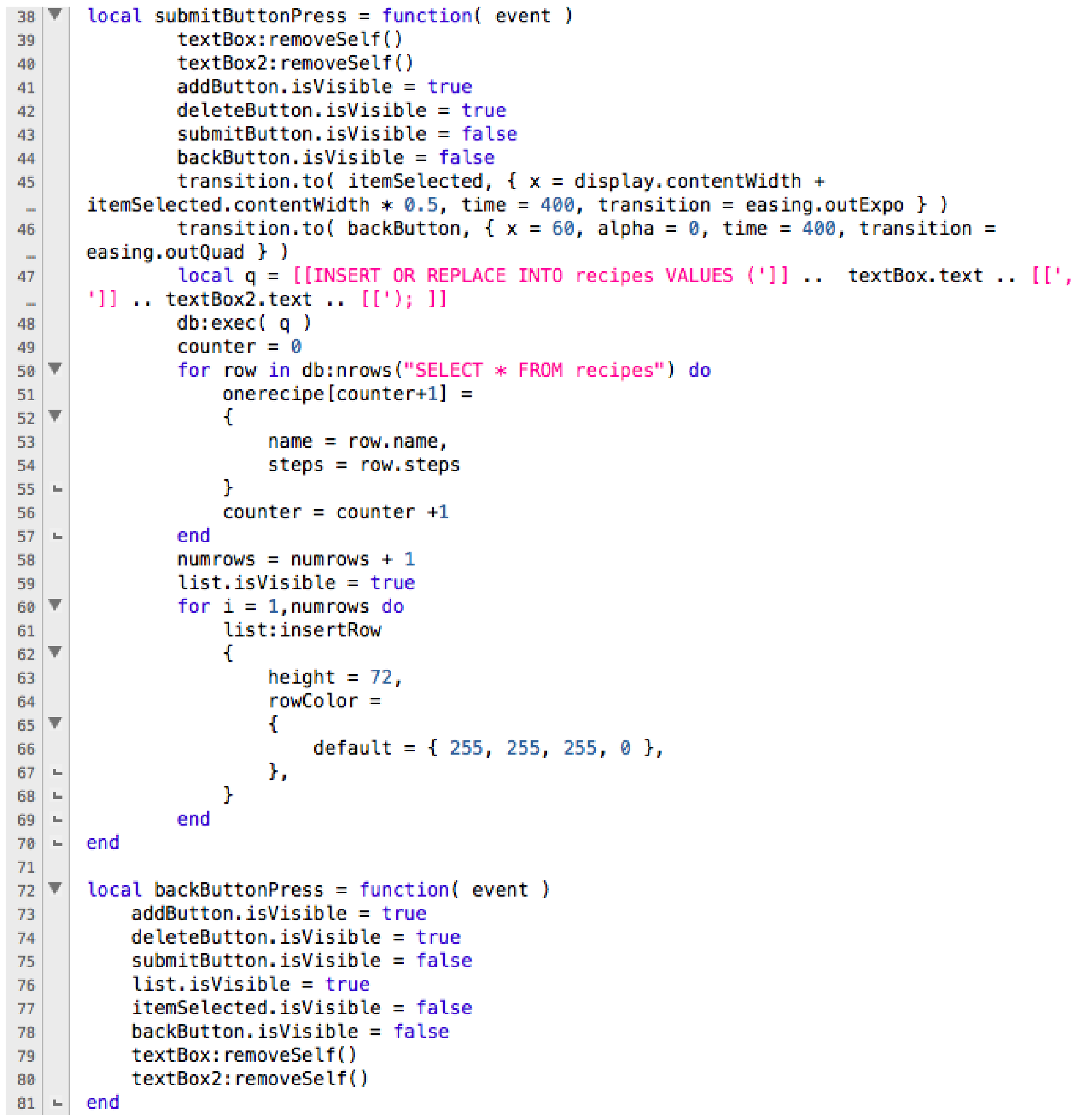
The end-user can add a new recipe and also delete any existing recipe by tapping the corresponding Add and Delete buttons. The touch events for the two buttons are implemented as follows (Figure 30):



```
local addButtonPress = function( event )
    local row = event.row
    local background = event.background
        textBox = native.newTextBox(display.screenOriginX, 100, display.contentWidth, 30)
        textBox.text = "NAME"--\nAnd this is included"
        textBox.isEditable = 'true'
        textBox.font = native.newFont( "Helvetica", 18 )
        textBox:setTextColor( 0, 0, 0, 255)
        textBox2 = native.newTextBox( display.screenOriginX, 150, display.contentWidth, 30
)
        textBox2.text = "STEPS \n\n"
        textBox2.isEditable = 'true'
        textBox2.font = native.newFont( "Helvetica", 18 )
        textBox2:setTextColor( 0, 0, 0, 255 )
        --Update the item selected text
        itemSelected.text = "Input Name and Steps"
        itemSelected.size = 20
        itemSelected.font = native.systemFont
        itemSelected:setTextColor(0,0,0)
        addButton.isVisible = false
        deleteButton.isVisible = false
        submitButton.isVisible = true
        transition.to( backButton, {alpha = 1, time = 400, transition = easing.outQuad} )
        backButton.isVisible = true
        list.isVisible = false
        transition.to( itemSelected, { x = display.contentCenterX, y = 200, time = 400,
transition = easing.outExpo } )
end

local deleteButtonPress = function( event )
        if deleteflag then
            deleteflag = false
            deleteButton:setLabel( "Delete Button" )
        else
            deleteflag = true
            deleteButton:setLabel( "Go Delete a Recipe" )
        end
end
```

Figure 22. Corona Productivity Application Code—Touch Events.

In the addButtonPress event handler function, two textbox widget objects are created and set to isEditable, while other onscreen widgets are set to not visible. The deleteButton event handler just changes the value of the flag used for onRowTouch(). The submitButtonPress function (in Figure we) updates the database table with the INSERT or REPLACE query, updates the onerecipe array, and then transitions the user back to the list screen.



```
local submitButtonPress = function( event )
        textBox:removeSelf()
        textBox2:removeSelf()
        addButton.isVisible = true
        deleteButton.isVisible = true
        submitButton.isVisible = false
        backButton.isVisible = false
        transition.to( itemSelected, { x = display.contentWidth + itemSelected.contentWidth * 0.5, time = 400, transition = easing.outExpo } )
        transition.to( backButton, { x = 60, alpha = 0, time = 400, transition = easing.outQuad } )
        local q = [[INSERT OR REPLACE INTO recipes VALUES (']] ..  textBox.text .. [[',']] .. textBox2.text .. [[');  ]]
        db:exec( q )
        counter = 0
        for row in db:nrows("SELECT * FROM recipes") do
            onerecipe[counter+1] =
            {
                name = row.name,
                steps = row.steps
            }
            counter = counter +1
        end
        numrows = numrows + 1
        list.isVisible = true
        for i = 1,numrows do
            list:insertRow
            {
                height = 72,
                rowColor =
                {
                    default = { 255, 255, 255, 0 },
                },
            }
        end
end

local backButtonPress = function( event )
    addButton.isVisible = true
    deleteButton.isVisible = true
    submitButton.isVisible = false
    list.isVisible = true
    itemSelected.isVisible = false
    backButton.isVisible = false
    textBox:removeSelf()
    textBox2:removeSelf()
end
```

Figure 23. Corona Productivity App Code—Touch Events 2.

Corona’s transition library was used on occasion when transitioning from one scene to another. It provides functions that animate objects during that time (e.g. changing positions or at a specified rate increasing or decreasing alpha, which is an object’s opacity).

2. Game Application

Figure 24 through 27 show the screens of the finished product on Android and iOS devices. Besides the platform-specific fonts and screen dimensions, they are visually the same.

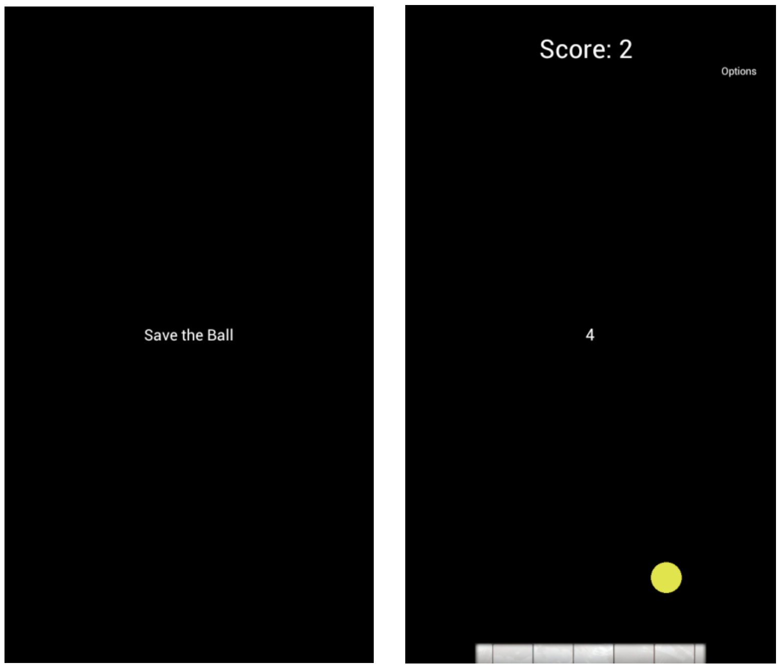


Figure 24. Corona Android Game Application.

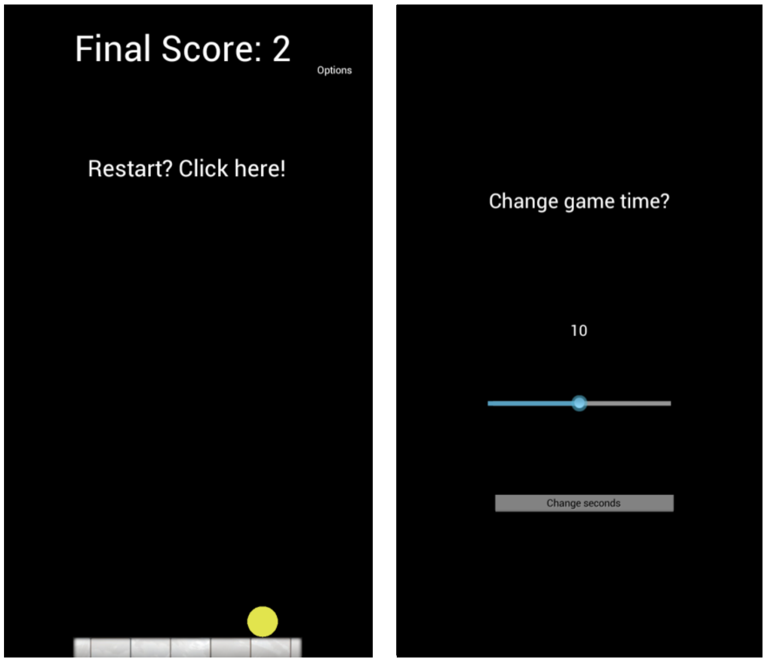


Figure 25.   Corona Android Game Application cont.

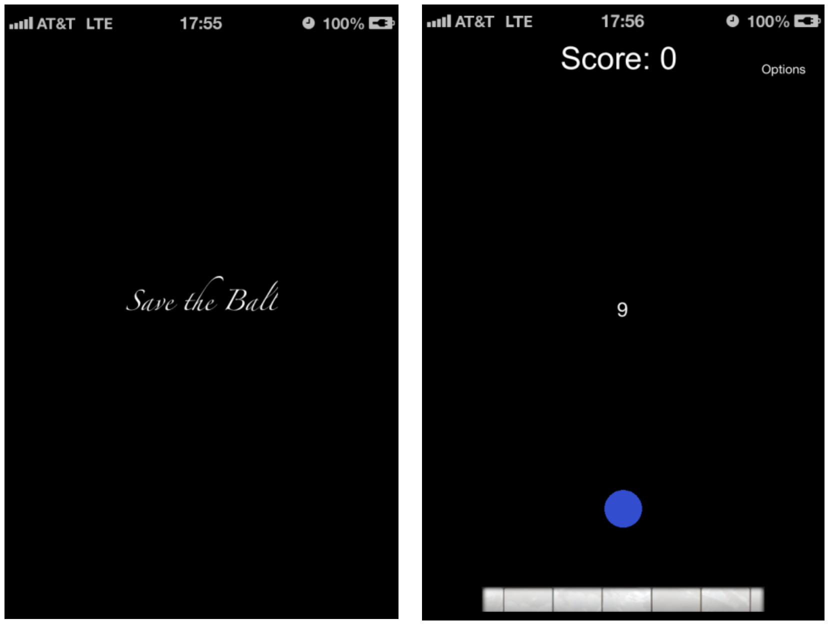


Figure 26.   Corona iOS Game Application.

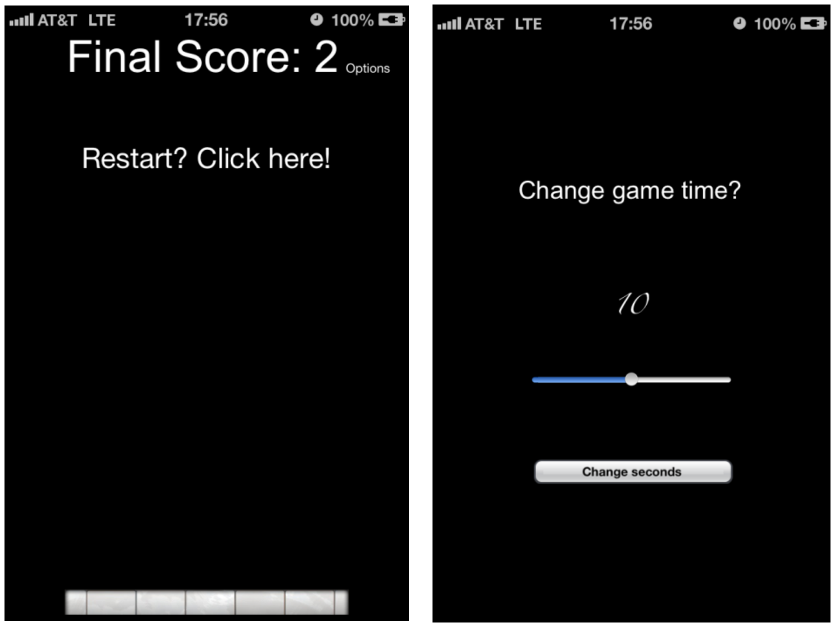
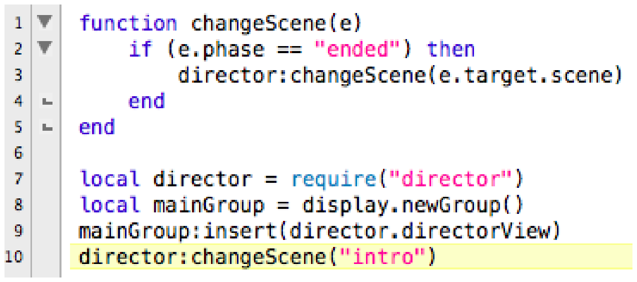


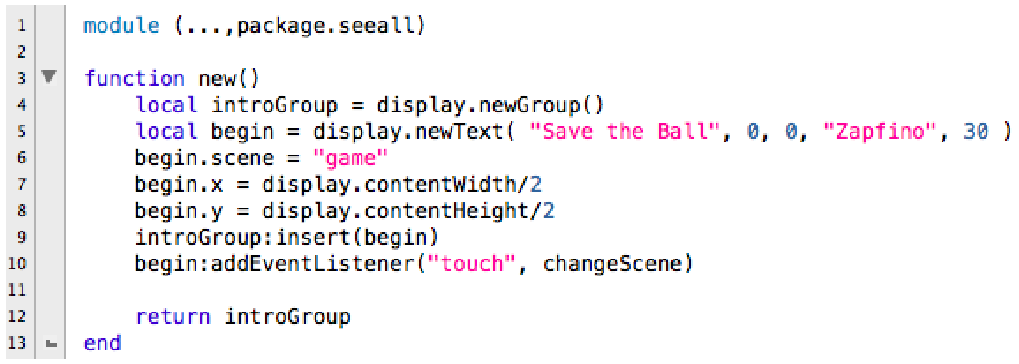
Figure 27. Corona iOS Game Application cont.



```
function changeScene(e)
    if (e.phase == "ended") then
        director:changeScene(e.target.scene)
    end
end

local director = require("director")
local mainGroup = display.newGroup()
mainGroup:insert(director.directorView)
director:changeScene("intro")
```

Figure 28. Corona Game Application Main.lua.

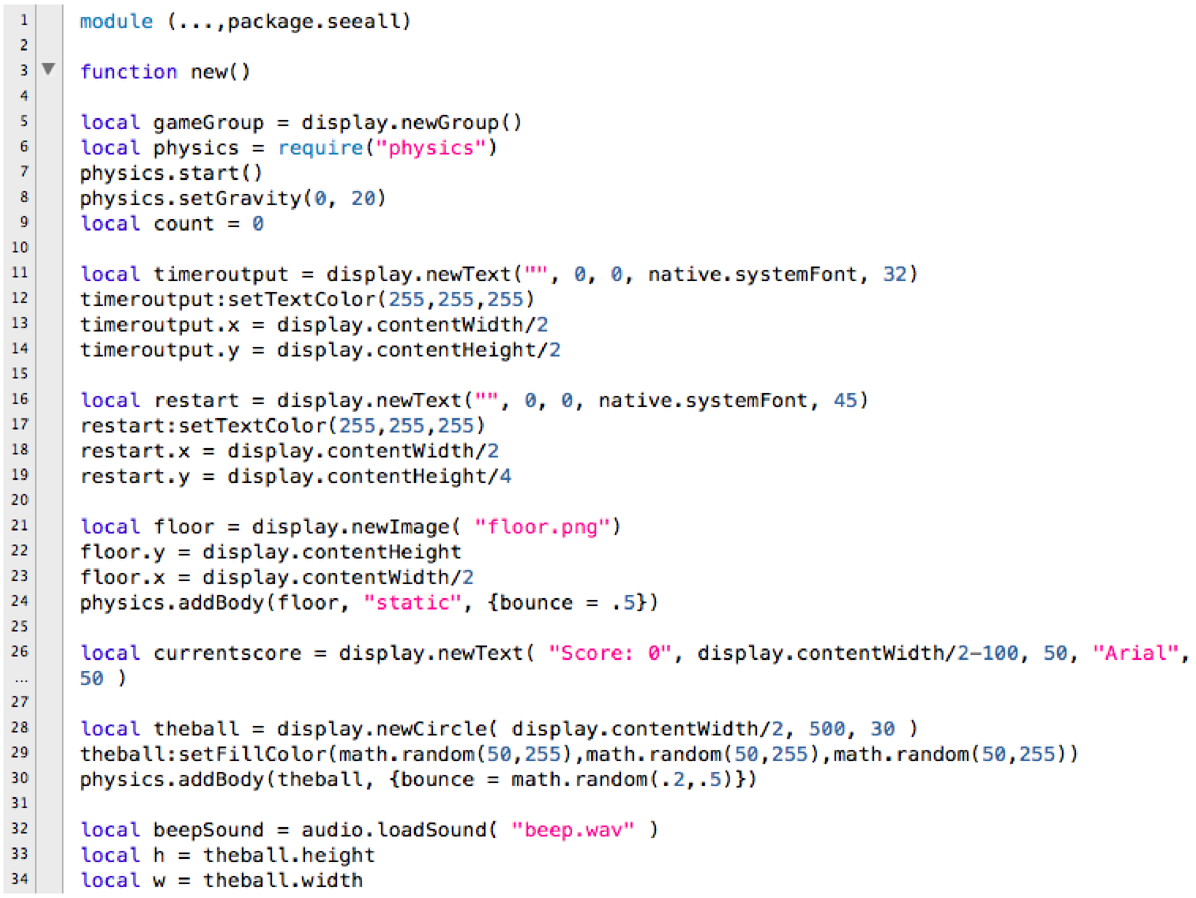
Figure 28 is a screenshot of main.lua, the first of four Lua files for our game application using Corona SDK. This file contains no predefined widgets. The UI of this application is mainly composed from graphic objects. We require the third-party-developed director class rather than the Corona storyboard class for scene changing in this particular application—this allows easy calling of new scenes to the display. Open source third party classes, like the rest of Corona’s community support, are numerous and can be found in Corona’s developer website. The director class’s changeScene function is used in the other Lua files with a screen object’s event listener, recognizing the new scene as **e.target.scene** [20]. Intro.lua, rather than main.lua, is the first scene that the user sees (Figure 29):



```
module (...,package.seeall)

function new()
    local introGroup = display.newGroup()
    local begin = display.newText( "Save the Ball", 0, 0, "Zapfino", 30 )
    begin.scene = "game"
    begin.x = display.contentWidth/2
    begin.y = display.contentHeight/2
    introGroup:insert(begin)
    begin:addEventListener("touch", changeScene)

    return introGroup
end
```

Figure 29. Corona Game Application Intro.lua.

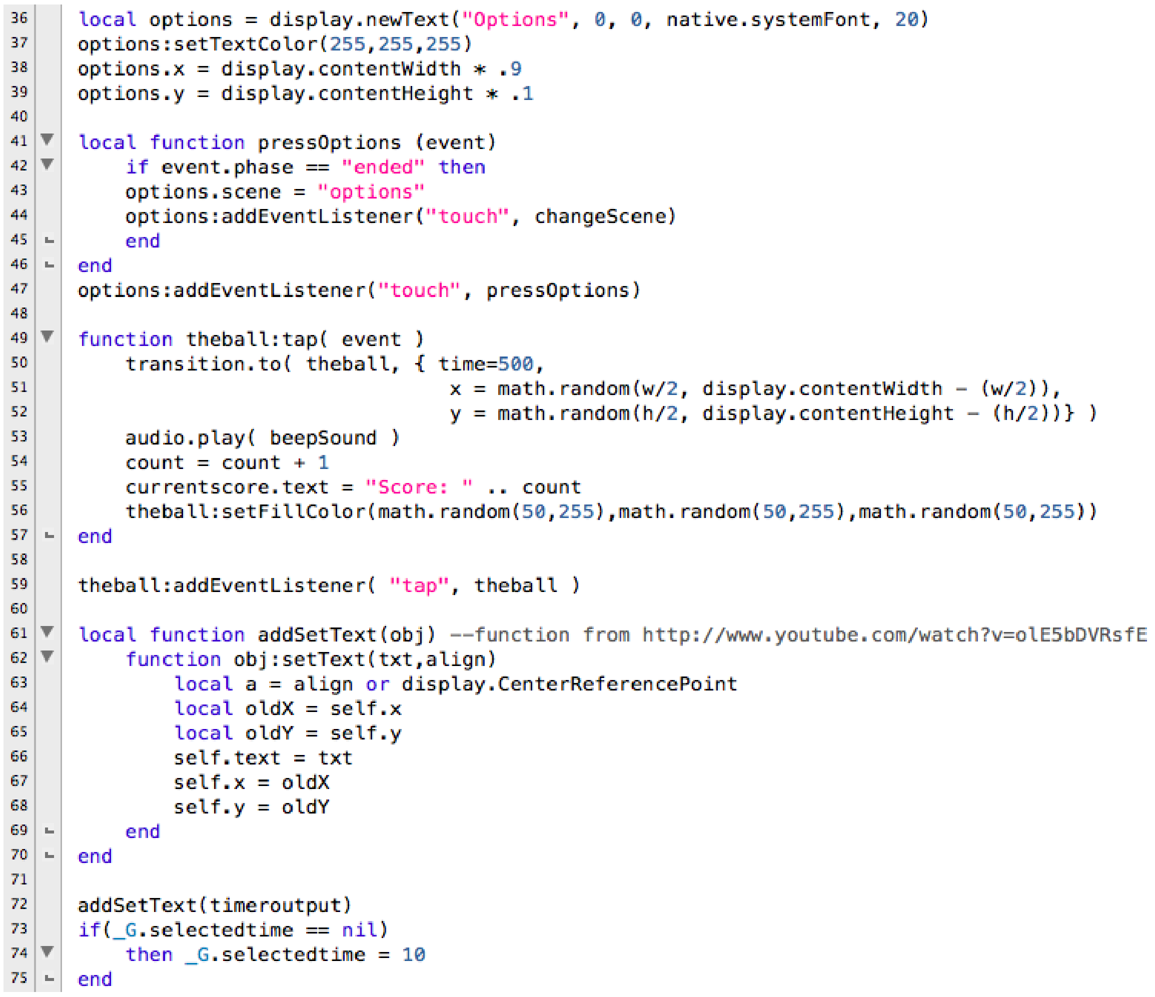
For files to utilize the director class, **module(…,package.seaall)** must be included so that seeall from the package in the project folder—which allows the use of main’s changescene()—can be used Also, the rest of the code must be enclosed in **function new()**. A text object is created with a touch event handler to take the user to the game screen (Figure 30).



```
module (...,package.seeall)

function new()

local gameGroup = display.newGroup()
local physics = require("physics")
physics.start()
physics.setGravity(0, 20)
local count = 0

local timeroutput = display.newText("", 0, 0, native.systemFont, 32)
timeroutput:setTextColor(255,255,255)
timeroutput.x = display.contentWidth/2
timeroutput.y = display.contentHeight/2

local restart = display.newText("", 0, 0, native.systemFont, 45)
restart:setTextColor(255,255,255)
restart.x = display.contentWidth/2
restart.y = display.contentHeight/4

local floor = display.newImage( "floor.png")
floor.y = display.contentHeight
floor.x = display.contentWidth/2
physics.addBody(floor, "static", {bounce = .5})

local currentscore = display.newText( "Score: 0", display.contentWidth/2-100, 50, "Arial",
50 )

local theball = display.newCircle( display.contentWidth/2, 500, 30 )
theball:setFillColor(math.random(50,255),math.random(50,255),math.random(50,255))
physics.addBody(theball, {bounce = math.random(.2,.5)})

local beepSound = audio.loadSound( "beep.wav" )
local h = theball.height
local w = theball.width
```

Figure 30. Corona Game Application Game.lua 1.

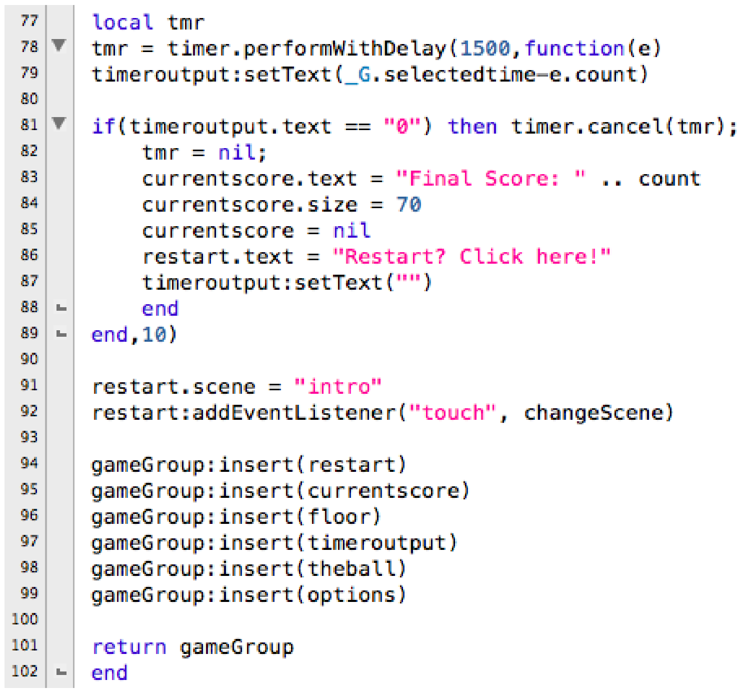
In game.lua, the physics library is required to utilize Corona's Box2d-like physics engine. Create on-scene UI objects **theball** and **floor** (which is a circle object and an image object, respectively) and add them as two bodies that have physics characteristics. The former is affected by gravity while the latter is static. Both have a bounce quality to it when physically colliding with another UI object.



```lua
local options = display.newText("Options", 0, 0, native.systemFont, 20)
options:setTextColor(255,255,255)
options.x = display.contentWidth * .9
options.y = display.contentHeight * .1

local function pressOptions (event)
    if event.phase == "ended" then
    options.scene = "options"
    options:addEventListener("touch", changeScene)
    end
end
options:addEventListener("touch", pressOptions)

function theball:tap( event )
    transition.to( theball, { time=500,
                               x = math.random(w/2, display.contentWidth - (w/2)),
                               y = math.random(h/2, display.contentHeight - (h/2))} )
    audio.play( beepSound )
    count = count + 1
    currentscore.text = "Score: " .. count
    theball:setFillColor(math.random(50,255),math.random(50,255),math.random(50,255))
end

theball:addEventListener( "tap", theball )

local function addSetText(obj) --function from http://www.youtube.com/watch?v=olE5bDVRsfE
    function obj:setText(txt,align)
        local a = align or display.CenterReferencePoint
        local oldX = self.x
        local oldY = self.y
        self.text = txt
        self.x = oldX
        self.y = oldY
    end
end

addSetText(timeroutput)
if(_G.selectedtime == nil)
    then _G.selectedtime = 10
end
```

Figure 31. Corona Game Application Game.lua 2.

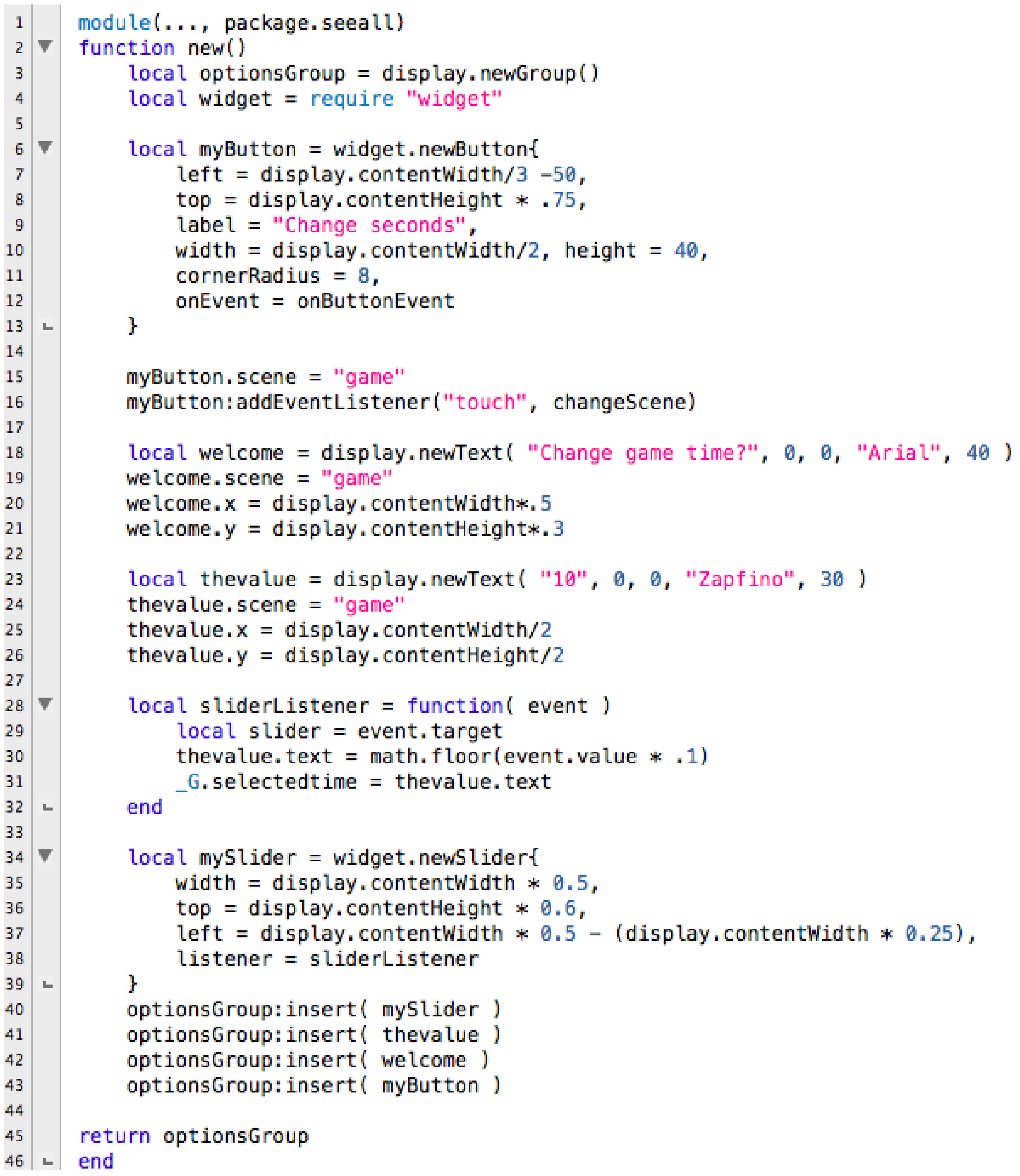
Game.lua continues in Figure 31, where there are three functions: **pressOptions(event)** is the event handler for the options text object pass the option scene to main as the scene to change to, **theball:tap(event)** utilizes the transition library to move the ball to a random screen position, and **addSetText()** helps setting and changing the position of text [21]. The shape object's setFillColor option here randomly changes the ball's color.



```
77   local tmr
78   tmr = timer.performWithDelay(1500,function(e)
79   timeroutput:setText(_G.selectedtime-e.count)
80
81   if(timeroutput.text == "0") then timer.cancel(tmr);
82       tmr = nil;
83       currentscore.text = "Final Score: " .. count
84       currentscore.size = 70
85       currentscore = nil
86       restart.text = "Restart? Click here!"
87       timeroutput:setText("")
88       end
89   end,10)
90
91   restart.scene = "intro"
92   restart:addEventListener("touch", changeScene)
93
94   gameGroup:insert(restart)
95   gameGroup:insert(currentscore)
96   gameGroup:insert(floor)
97   gameGroup:insert(timeroutput)
98   gameGroup:insert(theball)
99   gameGroup:insert(options)
100
101  return gameGroup
102  end
```

Figure 32. Corona Game Application Game.lua 3.

In Figure 32, the timer counts down once every 1500 milliseconds. Memory is deallocated for the timer object at line 85 and we give the restart text an event listener. Finally, on-scene objects are inserted into **gameGroup** and when that is returned all those objects are removed from the screen.



```lua
module(..., package.seeall)
function new()
    local optionsGroup = display.newGroup()
    local widget = require "widget"

    local myButton = widget.newButton{
        left = display.contentWidth/3 -50,
        top = display.contentHeight * .75,
        label = "Change seconds",
        width = display.contentWidth/2, height = 40,
        cornerRadius = 8,
        onEvent = onButtonEvent
    }

    myButton.scene = "game"
    myButton:addEventListener("touch", changeScene)

    local welcome = display.newText( "Change game time?", 0, 0, "Arial", 40 )
    welcome.scene = "game"
    welcome.x = display.contentWidth*.5
    welcome.y = display.contentHeight*.3

    local thevalue = display.newText( "10", 0, 0, "Zapfino", 30 )
    thevalue.scene = "game"
    thevalue.x = display.contentWidth/2
    thevalue.y = display.contentHeight/2

    local sliderListener = function( event )
        local slider = event.target
        thevalue.text = math.floor(event.value * .1)
        _G.selectedtime = thevalue.text
    end

    local mySlider = widget.newSlider{
        width = display.contentWidth * 0.5,
        top = display.contentHeight * 0.6,
        left = display.contentWidth * 0.5 - (display.contentWidth * 0.25),
        listener = sliderListener
    }
    optionsGroup:insert( mySlider )
    optionsGroup:insert( thevalue )
    optionsGroup:insert( welcome )
    optionsGroup:insert( myButton )

return optionsGroup
end
```

Figure 33.   Corona Game Application Options.lua.

Figure 33 shows options.lua. We create two widget objects, a button and a slider. The button object has an event listener for a director class scene change just some of the aforementioned objects do. The sliderListener updates the global variable **_G** to a time between 0 and 10.

3. Device-Accessing Application

Figures 34 through 37 show the screens of the finished product on Android and iOS devices. Besides the platform-specific maps used, they are visually the same.

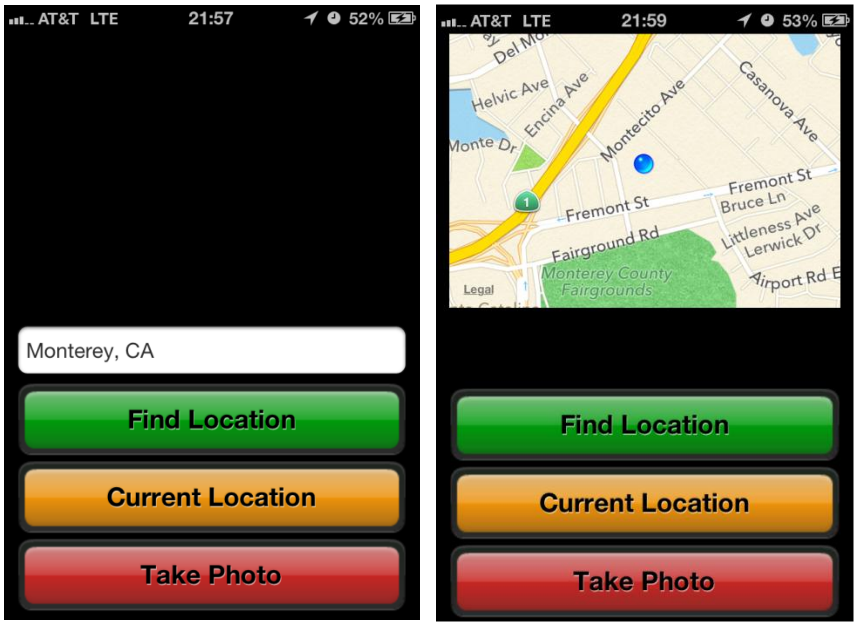


Figure 34. Corona iOS Device-Accessing Application.

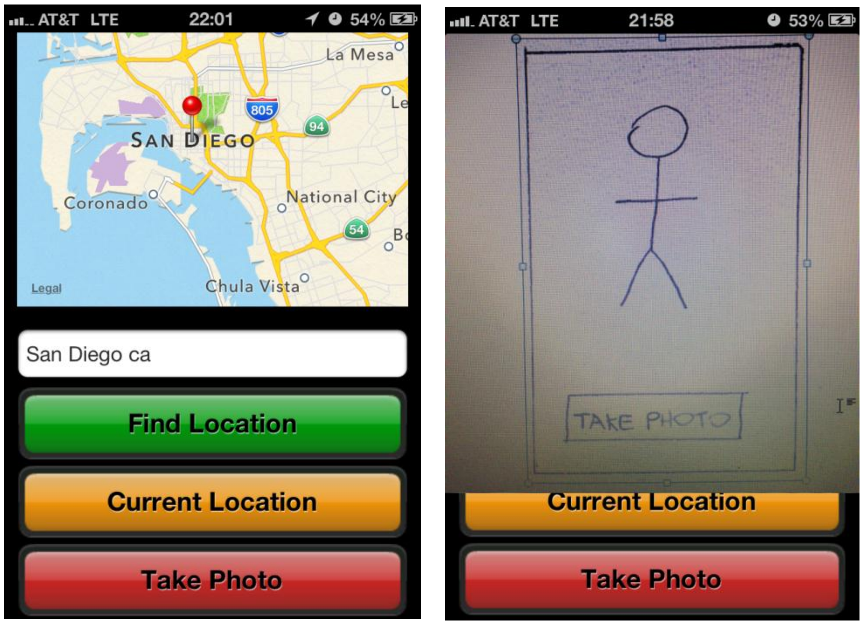


Figure 35. Corona iOS Device-Accessing Application cont.

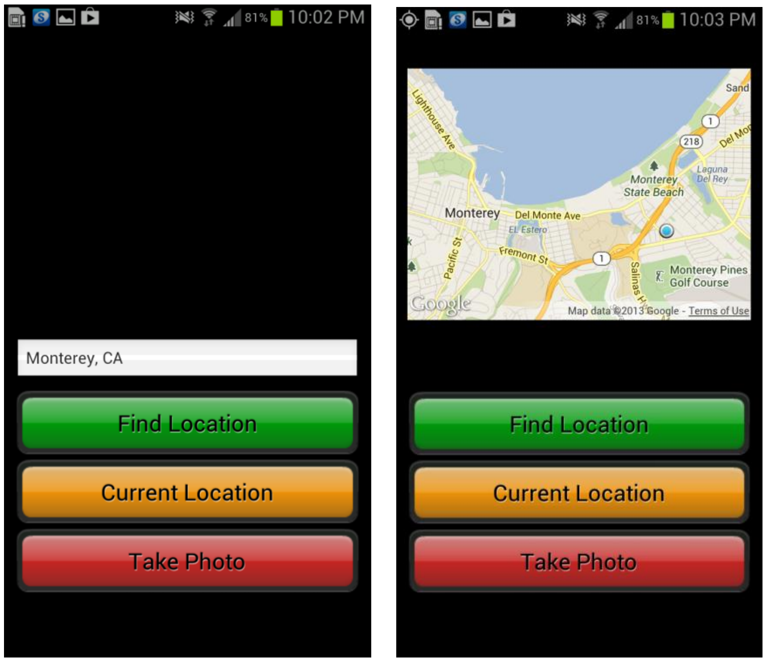
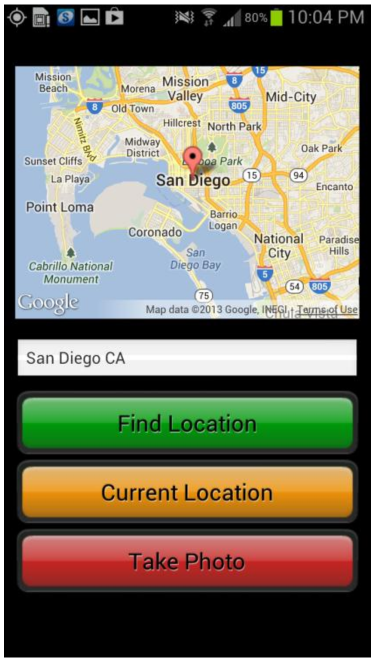


Figure 36. Corona Android Device-Accessing Application.



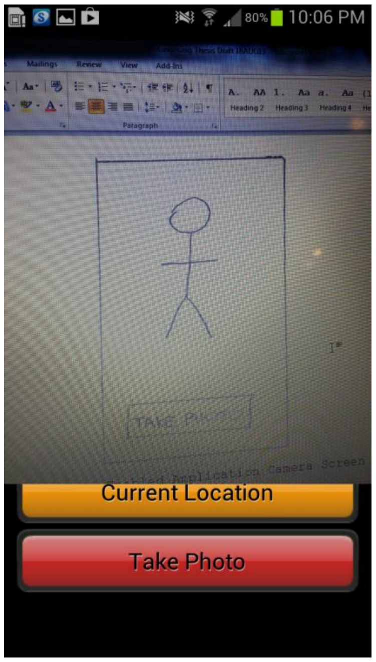
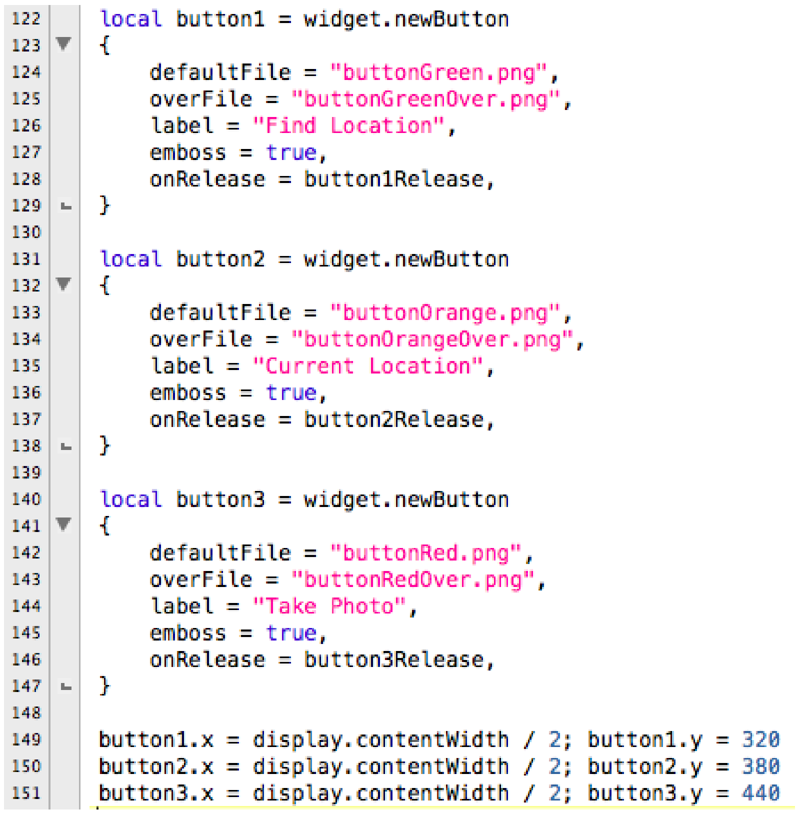


Figure 37. Corona Android Device-Accessing Application cont.

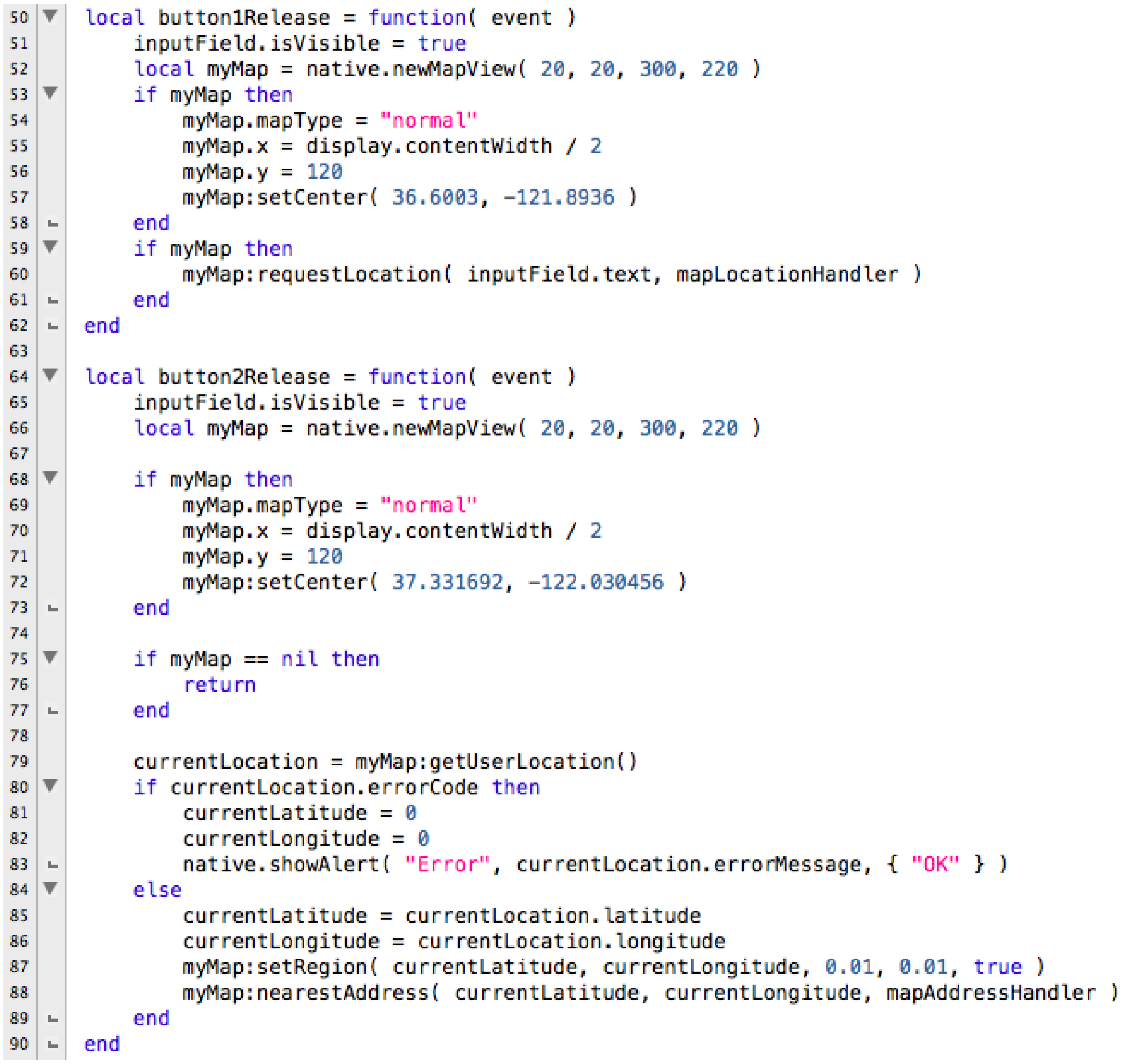
The following four figures (38 through 41) are screenshots of our code. They include UI object creation for our buttons, map, and textbox, as well as event-handling for accessing the device’s camera and geolocation.



```
local button1 = widget.newButton
{
    defaultFile = "buttonGreen.png",
    overFile = "buttonGreenOver.png",
    label = "Find Location",
    emboss = true,
    onRelease = button1Release,
}

local button2 = widget.newButton
{
    defaultFile = "buttonOrange.png",
    overFile = "buttonOrangeOver.png",
    label = "Current Location",
    emboss = true,
    onRelease = button2Release,
}

local button3 = widget.newButton
{
    defaultFile = "buttonRed.png",
    overFile = "buttonRedOver.png",
    label = "Take Photo",
    emboss = true,
    onRelease = button3Release,
}

button1.x = display.contentWidth / 2; button1.y = 320
button2.x = display.contentWidth / 2; button2.y = 380
button3.x = display.contentWidth / 2; button3.y = 440
```

Figure 38. Corona Device-Accessing Application.

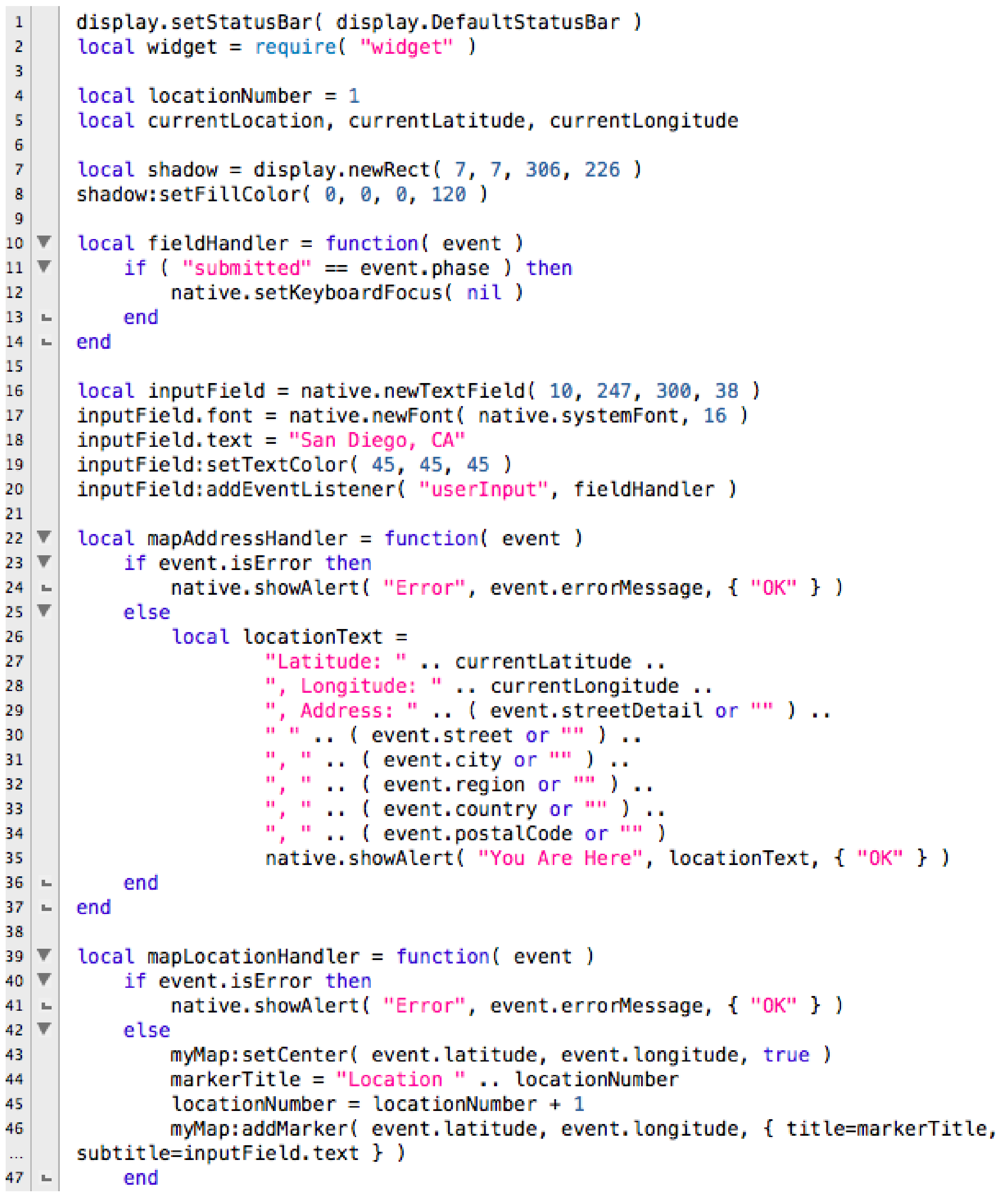
Figure 38 shows how we create and then add on screen the button widgets, which are images. The mapView is created later within the map button event-handlers (Figure 39). By including Corona's Widgets API, few lines of code are necessary to create and then position a map.



```lua
local button1Release = function( event )
    inputField.isVisible = true
    local myMap = native.newMapView( 20, 20, 300, 220 )
    if myMap then
        myMap.mapType = "normal"
        myMap.x = display.contentWidth / 2
        myMap.y = 120
        myMap:setCenter( 36.6003, -121.8936 )
    end
    if myMap then
        myMap:requestLocation( inputField.text, mapLocationHandler )
    end
end

local button2Release = function( event )
    inputField.isVisible = true
    local myMap = native.newMapView( 20, 20, 300, 220 )

    if myMap then
        myMap.mapType = "normal"
        myMap.x = display.contentWidth / 2
        myMap.y = 120
        myMap:setCenter( 37.331692, -122.030456 )
    end

    if myMap == nil then
        return
    end

    currentLocation = myMap:getUserLocation()
    if currentLocation.errorCode then
        currentLatitude = 0
        currentLongitude = 0
        native.showAlert( "Error", currentLocation.errorMessage, { "OK" } )
    else
        currentLatitude = currentLocation.latitude
        currentLongitude = currentLocation.longitude
        myMap:setRegion( currentLatitude, currentLongitude, 0.01, 0.01, true )
        myMap:nearestAddress( currentLatitude, currentLongitude, mapAddressHandler )
    end
end
```

Figure 39.   Corona Device-Accessing Application cont.

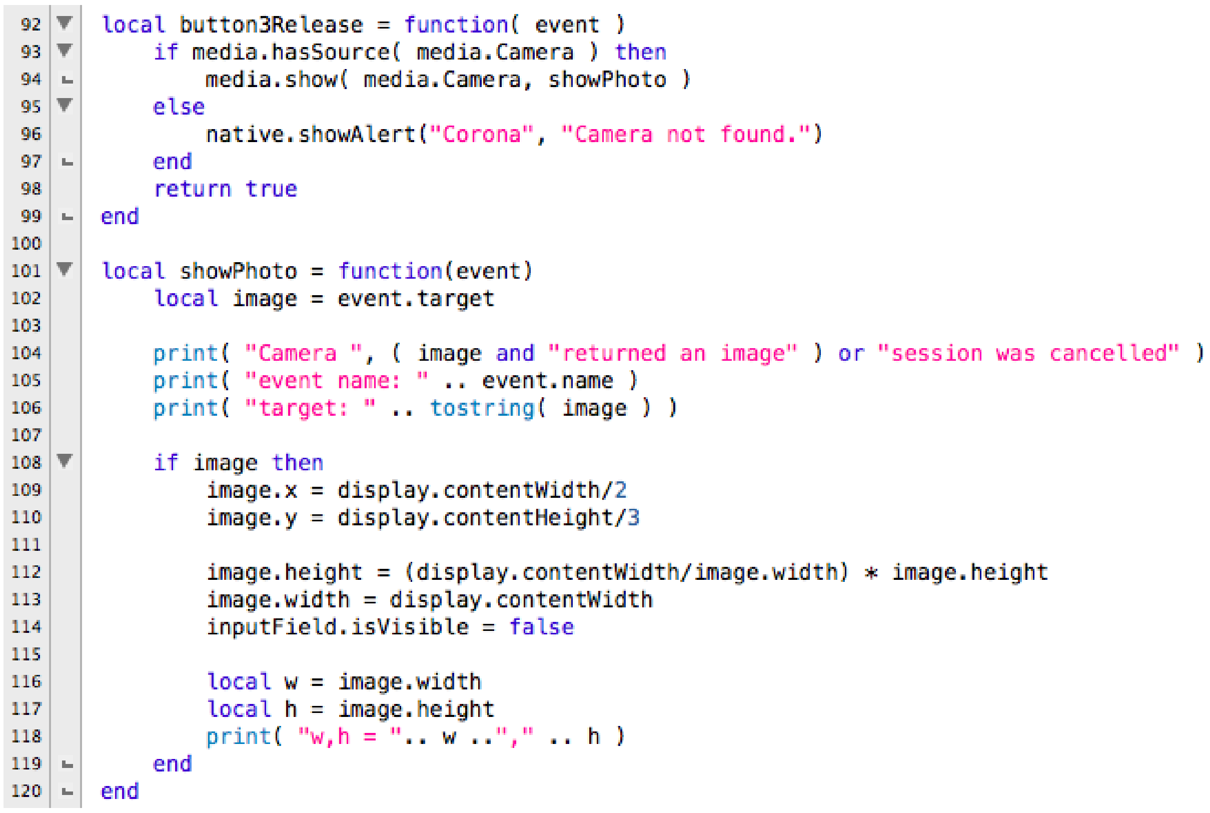
button1Release() uses the Map widget's **requestLocation** function to take text from the **inputField** and find its location on the map via **mapLocationHandler** (Figure 40). **button2Release()** initializes the map to a given location, fetches the device's current location using **getUserLocation()**, and then centers the map on that location.



```lua
display.setStatusBar( display.DefaultStatusBar )
local widget = require( "widget" )

local locationNumber = 1
local currentLocation, currentLatitude, currentLongitude

local shadow = display.newRect( 7, 7, 306, 226 )
shadow:setFillColor( 0, 0, 0, 120 )

local fieldHandler = function( event )
    if ( "submitted" == event.phase ) then
        native.setKeyboardFocus( nil )
    end
end

local inputField = native.newTextField( 10, 247, 300, 38 )
inputField.font = native.newFont( native.systemFont, 16 )
inputField.text = "San Diego, CA"
inputField:setTextColor( 45, 45, 45 )
inputField:addEventListener( "userInput", fieldHandler )

local mapAddressHandler = function( event )
    if event.isError then
        native.showAlert( "Error", event.errorMessage, { "OK" } )
    else
        local locationText =
                "Latitude: " .. currentLatitude ..
                ", Longitude: " .. currentLongitude ..
                ", Address: " .. ( event.streetDetail or "" ) ..
                " " .. ( event.street or "" ) ..
                ", " .. ( event.city or "" ) ..
                ", " .. ( event.region or "" ) ..
                ", " .. ( event.country or "" ) ..
                ", " .. ( event.postalCode or "" )
                native.showAlert( "You Are Here", locationText, { "OK" } )
    end
end

local mapLocationHandler = function( event )
    if event.isError then
        native.showAlert( "Error", event.errorMessage, { "OK" } )
    else
        myMap:setCenter( event.latitude, event.longitude, true )
        markerTitle = "Location " .. locationNumber
        locationNumber = locationNumber + 1
        myMap:addMarker( event.latitude, event.longitude, { title=markerTitle, subtitle=inputField.text } )
    end
```

Figure 40.   Corona Device-Accessing Application 3.

mapAddressHandler() displays an alert with the current location, which uses Map's **nearestAddress()** method to find the nearest address based on the given longitude and latitude. **mapLocationHandler()** centers the map on the coordinates and adds a pin to the map representing the location.



```
local button3Release = function( event )
    if media.hasSource( media.Camera ) then
        media.show( media.Camera, showPhoto )
    else
        native.showAlert("Corona", "Camera not found.")
    end
    return true
end

local showPhoto = function(event)
    local image = event.target

    print( "Camera ", ( image and "returned an image" ) or "session was cancelled" )
    print( "event name: " .. event.name )
    print( "target: " .. tostring( image ) )

    if image then
        image.x = display.contentWidth/2
        image.y = display.contentHeight/3

        image.height = (display.contentWidth/image.width) * image.height
        image.width = display.contentWidth
        inputField.isVisible = false

        local w = image.width
        local h = image.height
        print( "w,h = ".. w ..",".. h )
    end
end
```

Figure 41. Corona Device-Accessing Application 4.

Figure 41 describes how to access the device's camera. In **button3Release()**, if available the **media.show** function will open the interface to the camera (and photo library as well). **showPhoto()** is the listener that handles the returned image and positions and sizes the image on screen.

B. TITANIUM

1. Productivity Application

Figures 42 through 44 show the screens of the finished product on Android and iOS devices. Note the placement of the add and delete buttons on each platform; with one codebase, we utilize conditional branching in our code to access some native button aspects depending on the platform in use.

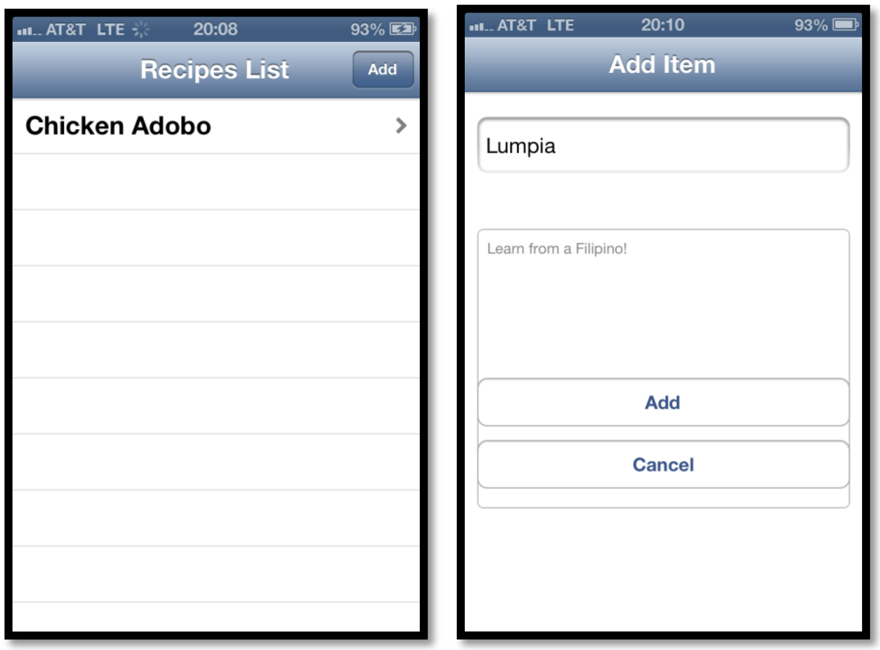


Figure 42. Titanium iOS Productivity Application.

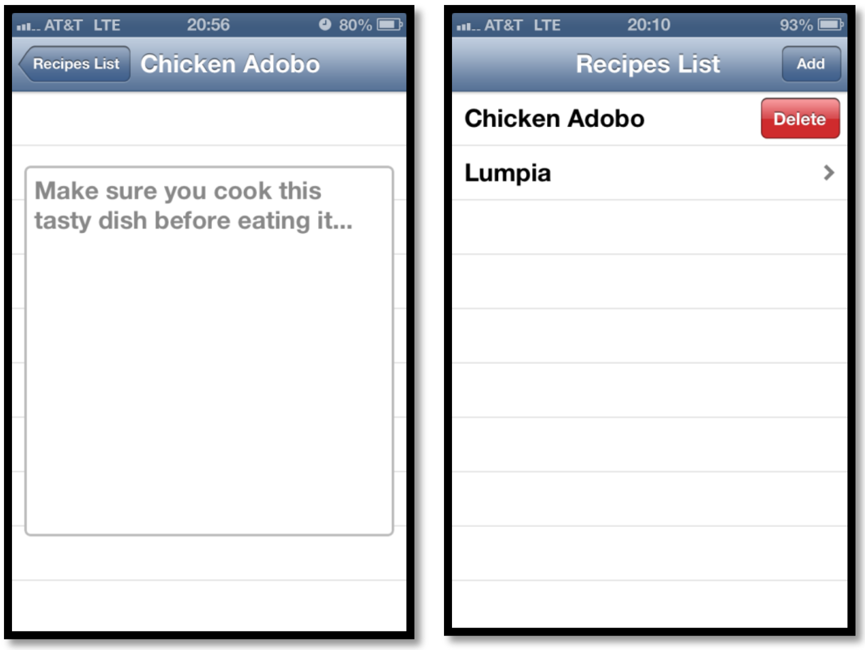


Figure 43. Titanium iOS Productivity Application cont.

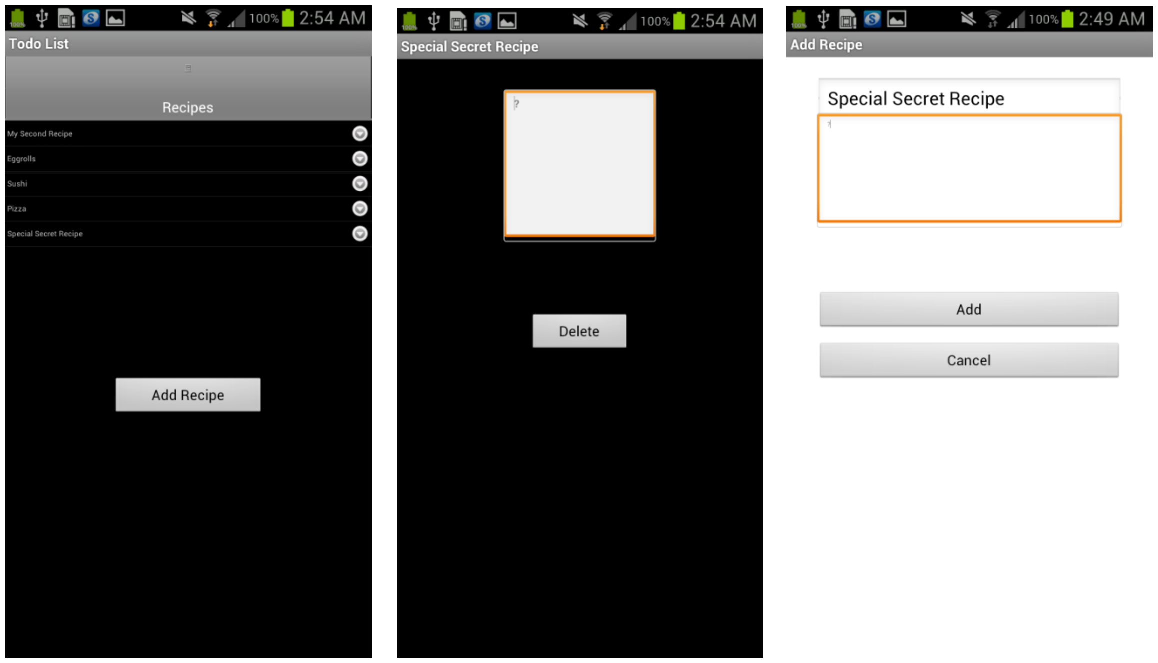
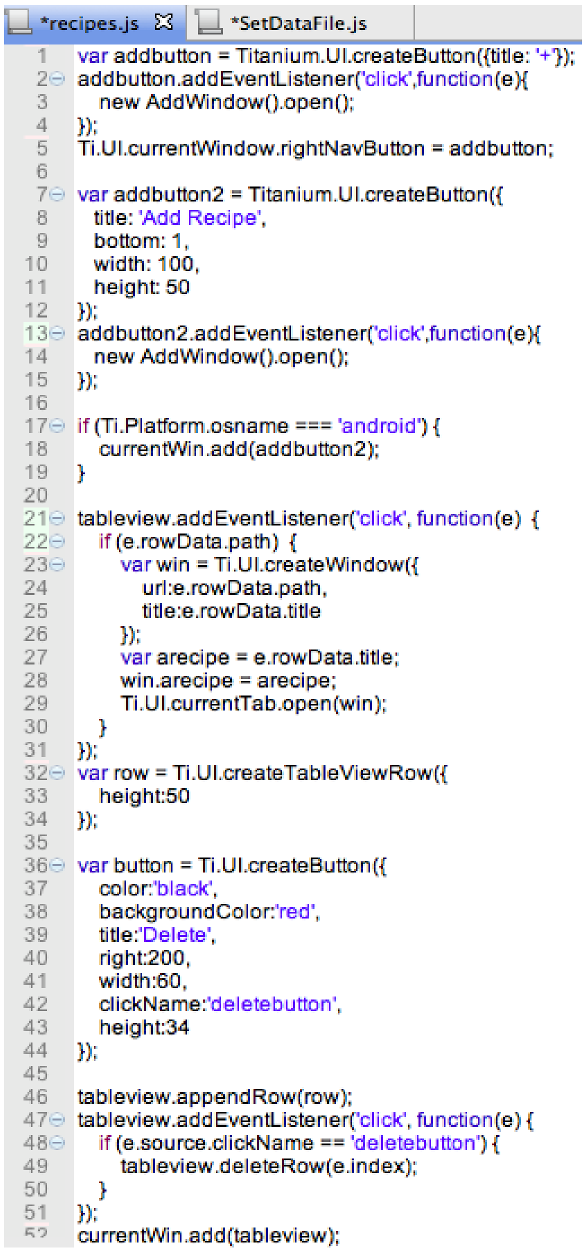


Figure 44. Titanium Android Productivity Application.

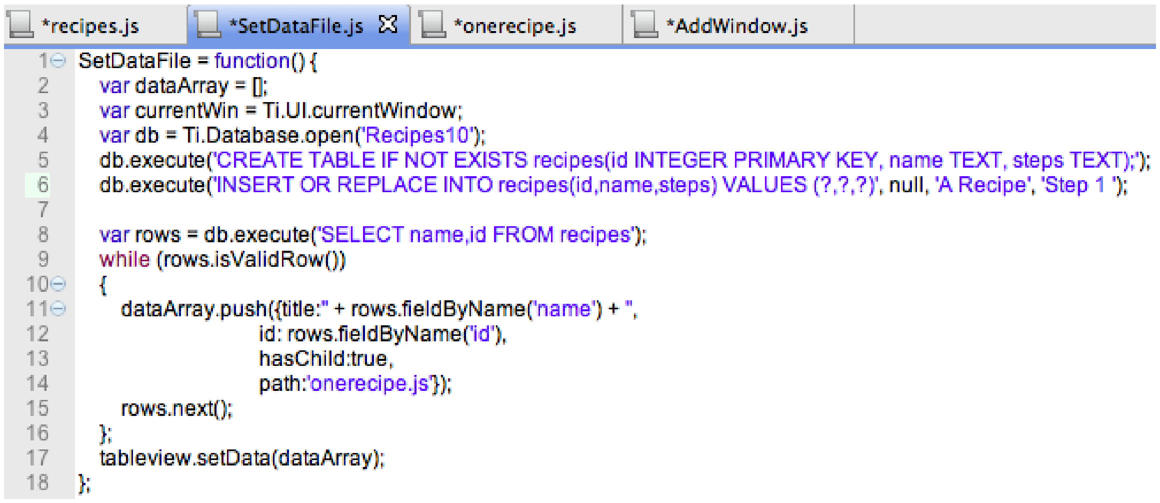
In the main window, there is a list (table view) and an Add button. Figure 45 is a screenshot of Recipes.js, which includes JavaScript statements to construct this window. We create a window, add the table view, retrieve the data from the database by row, and create an add button based on the platform. The “Add Recipe” button is only added to the window when the application is on an Android device. The “+” button is right-of-title navigation button only visible on iOS devices.



```
var addbutton = Titanium.UI.createButton({title: '+'});
addbutton.addEventListener('click',function(e){
   new AddWindow().open();
});
Ti.UI.currentWindow.rightNavButton = addbutton;

var addbutton2 = Titanium.UI.createButton({
  title: 'Add Recipe',
  bottom: 1,
  width: 100,
  height: 50
});
addbutton2.addEventListener('click',function(e){
  new AddWindow().open();
});

if (Ti.Platform.osname === 'android') {
   currentWin.add(addbutton2);
}

tableview.addEventListener('click', function(e) {
   if (e.rowData.path) {
      var win = Ti.UI.createWindow({
         url:e.rowData.path,
         title:e.rowData.title
      });
      var arecipe = e.rowData.title;
      win.arecipe = arecipe;
      Ti.UI.currentTab.open(win);
   }
});
var row = Ti.UI.createTableViewRow({
   height:50
});

var button = Ti.UI.createButton({
   color:'black',
   backgroundColor:'red',
   title:'Delete',
   right:200,
   width:60,
   clickName:'deletebutton',
   height:34
});

tableview.appendRow(row);
tableview.addEventListener('click', function(e) {
   if (e.source.clickName == 'deletebutton') {
      tableview.deleteRow(e.index);
   }
});
currentWin.add(tableview);
```

Figure 45. Titanium Productivity Application Code.

When an item in the list is selected (by the end-user tapping on it), the second window is displayed to show the details of the selected item. This transition from one window to another is achieved in SetDataFile.js (Figure 46). When filling in each row of the list view with database entries, we set the path to onerecipe.js (Figure 47), where details of the chosen recipe are coded to be displayed.



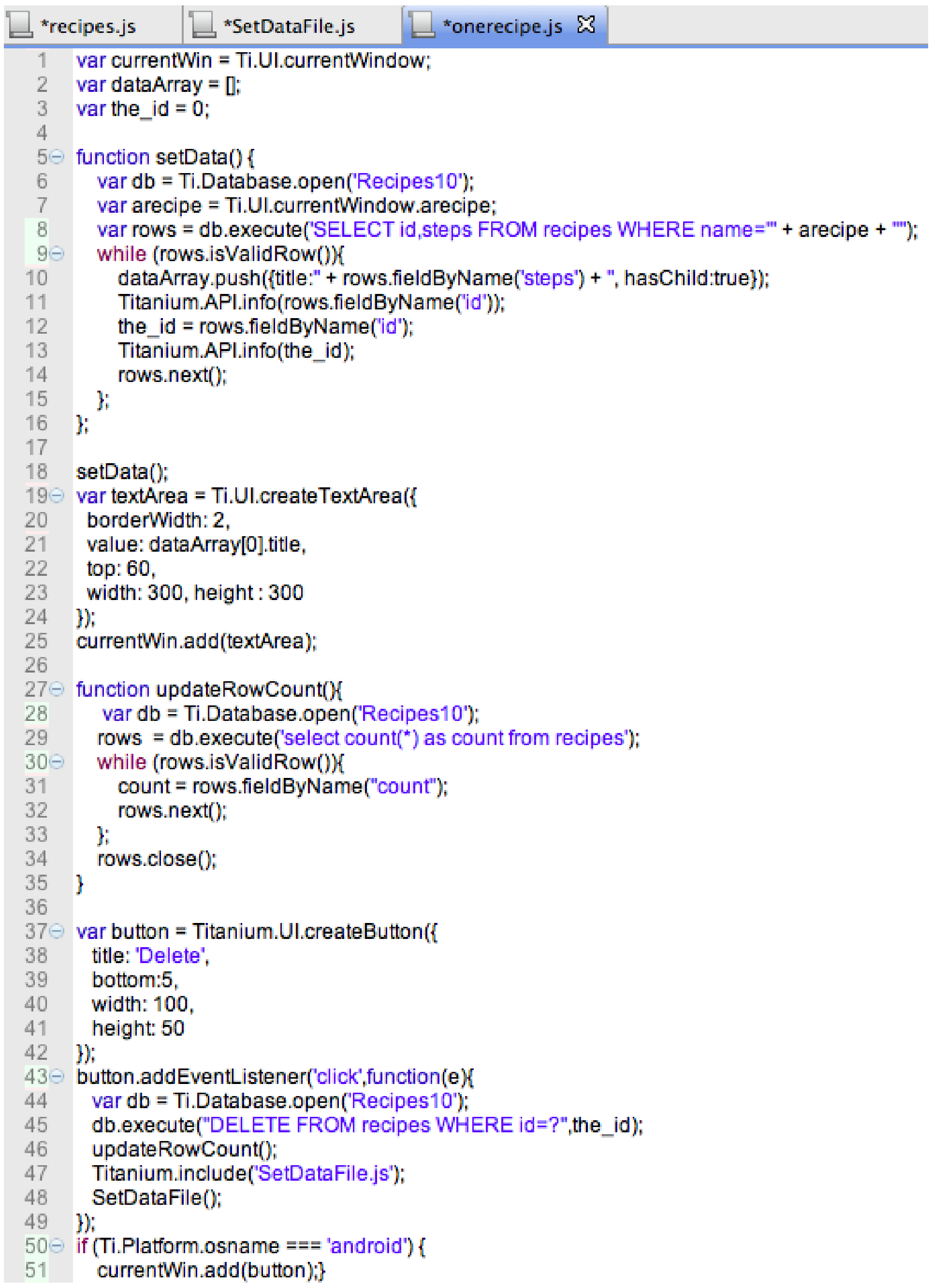
*recipes.js | *SetDataFile.js | *onerecipe.js | *AddWindow.js

```
SetDataFile = function() {
    var dataArray = [];
    var currentWin = Ti.UI.currentWindow;
    var db = Ti.Database.open('Recipes10');
    db.execute('CREATE TABLE IF NOT EXISTS recipes(id INTEGER PRIMARY KEY, name TEXT, steps TEXT);');
    db.execute('INSERT OR REPLACE INTO recipes(id,name,steps) VALUES (?,?,?)', null, 'A Recipe', 'Step 1 ');

    var rows = db.execute('SELECT name,id FROM recipes');
    while (rows.isValidRow())
    {
        dataArray.push({title:'' + rows.fieldByName('name') + '',
                        id: rows.fieldByName('id'),
                        hasChild:true,
                        path:'onerecipe.js'});
        rows.next();
    };
    tableview.setData(dataArray);
};
```

Figure 46.   Titanium Productivity Application Code.

As shown in Figure 46, the backend database in Titanium is managed in a manner analogous to the one in Corona.

In onerecipe.js (Figure 47) we create a delete button and add it to the recipe steps window only when on an Android device. We also update the row count of the database so that it will continue to be properly displayed in the list view screen [19].

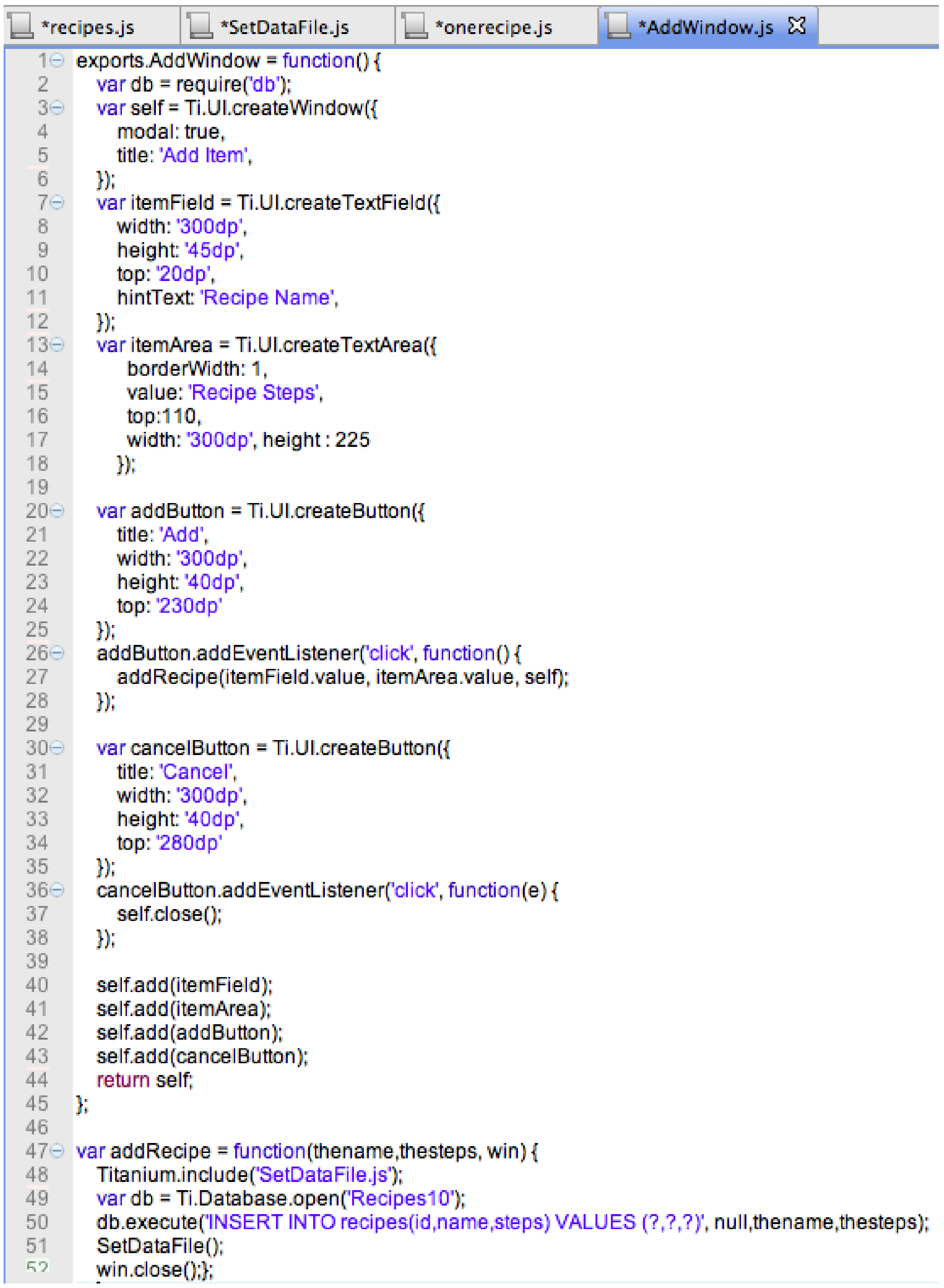


```
var currentWin = Ti.UI.currentWindow;
var dataArray = [];
var the_id = 0;

function setData() {
   var db = Ti.Database.open('Recipes10');
   var arecipe = Ti.UI.currentWindow.arecipe;
   var rows = db.execute('SELECT id,steps FROM recipes WHERE name="' + arecipe + '"');
   while (rows.isValidRow()){
      dataArray.push({title:'' + rows.fieldByName('steps') + '', hasChild:true});
      Titanium.API.info(rows.fieldByName('id'));
      the_id = rows.fieldByName('id');
      Titanium.API.info(the_id);
      rows.next();
   };
};

setData();
var textArea = Ti.UI.createTextArea({
  borderWidth: 2,
  value: dataArray[0].title,
  top: 60,
  width: 300, height : 300
});
currentWin.add(textArea);

function updateRowCount(){
    var db = Ti.Database.open('Recipes10');
   rows  = db.execute('select count(*) as count from recipes');
   while (rows.isValidRow()){
      count = rows.fieldByName("count");
      rows.next();
   };
   rows.close();
}

var button = Titanium.UI.createButton({
   title: 'Delete',
   bottom:5,
   width: 100,
   height: 50
});
button.addEventListener('click',function(e){
   var db = Ti.Database.open('Recipes10');
   db.execute("DELETE FROM recipes WHERE id=?",the_id);
   updateRowCount();
   Titanium.include('SetDataFile.js');
   SetDataFile();
});
if (Ti.Platform.osname === 'android') {
   currentWin.add(button);}
```

Figure 47.   Titanium Productivity Application Code.

AddWindow() is the function that leads to the screen that add recipes. The event for the add button addRecipe() is implemented in Figure 48, where we use a basic SQL insert command and then update the rows in the list screen to match the updated database.

The delete button event handlers for the application on iOS and Android platforms are implemented on Figures 45 and 47, respectively, where we use a basic SQL delete command and then also update the rows in the list screen to match the updated database.



*recipes.js | *SetDataFile.js | *onerecipe.js | *AddWindow.js

```
exports.AddWindow = function() {
  var db = require('db');
  var self = Ti.UI.createWindow({
    modal: true,
    title: 'Add Item',
  });
  var itemField = Ti.UI.createTextField({
    width: '300dp',
    height: '45dp',
    top: '20dp',
    hintText: 'Recipe Name',
  });
  var itemArea = Ti.UI.createTextArea({
     borderWidth: 1,
     value: 'Recipe Steps',
     top:110,
     width: '300dp', height : 225
    });

  var addButton = Ti.UI.createButton({
    title: 'Add',
    width: '300dp',
    height: '40dp',
    top: '230dp'
  });
  addButton.addEventListener('click', function() {
    addRecipe(itemField.value, itemArea.value, self);
  });

  var cancelButton = Ti.UI.createButton({
    title: 'Cancel',
    width: '300dp',
    height: '40dp',
    top: '280dp'
  });
  cancelButton.addEventListener('click', function(e) {
    self.close();
  });

  self.add(itemField);
  self.add(itemArea);
  self.add(addButton);
  self.add(cancelButton);
  return self;
};

var addRecipe = function(thename,thesteps, win) {
  Titanium.include('SetDataFile.js');
  var db = Ti.Database.open('Recipes10');
  db.execute('INSERT INTO recipes(id,name,steps) VALUES (?,?,?)', null,thename,thesteps);
  SetDataFile();
  win.close();};
```

Figure 48.   Titanium Productivity Application Code.

2. Game Application

Until September 2013, there was no complete Titanium tool or library to conduct mobile game development [22]. Though an open source Box2d module was introduced by Appcelerator back in September 2011, it supported only a limited number of Box2d APIs, had little documentation, had not been improved upon in a long time, and could only be used on a single platform—iOS [23].

Appcelerator is now funding a mobile gaming startup company called Lanica, whose co-founder was also the co-founder of Corona SDK. Lanica has "developed a JavaScript-based tool set called Platino that will allow Titanium SDK developers to create high-performance games" [24]. Due to the limited time we had to work with Lanica, much of the code we use is adopted from Lanica's sample code and documentation [22], and we also did not include code for either a home screen or an option screen. Figure 49 is a screenshot of our Titanium-Platino game application.

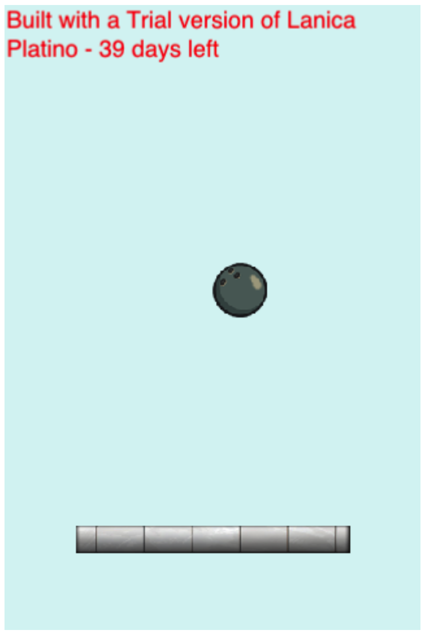
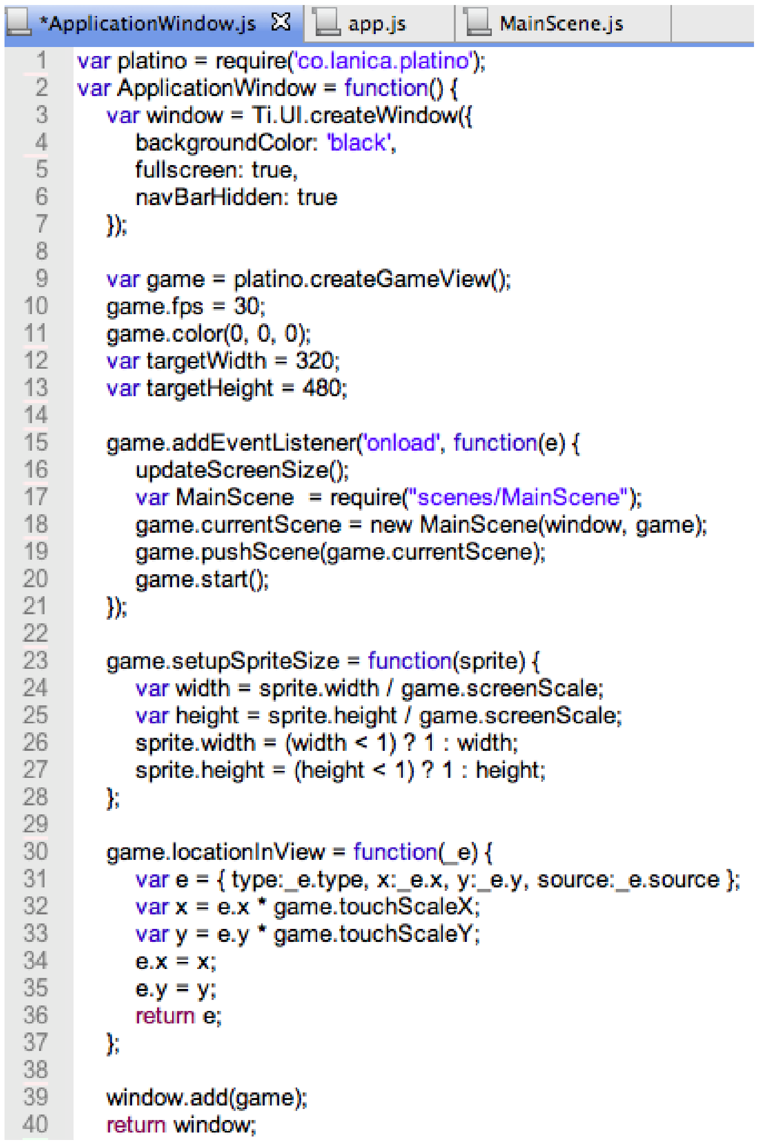


Figure 49. Titanium Game Application

The user interface consists of the game screen, the ball object, and the floor object. Figure 50 and 51 are the JavaScript statements to construct this window.

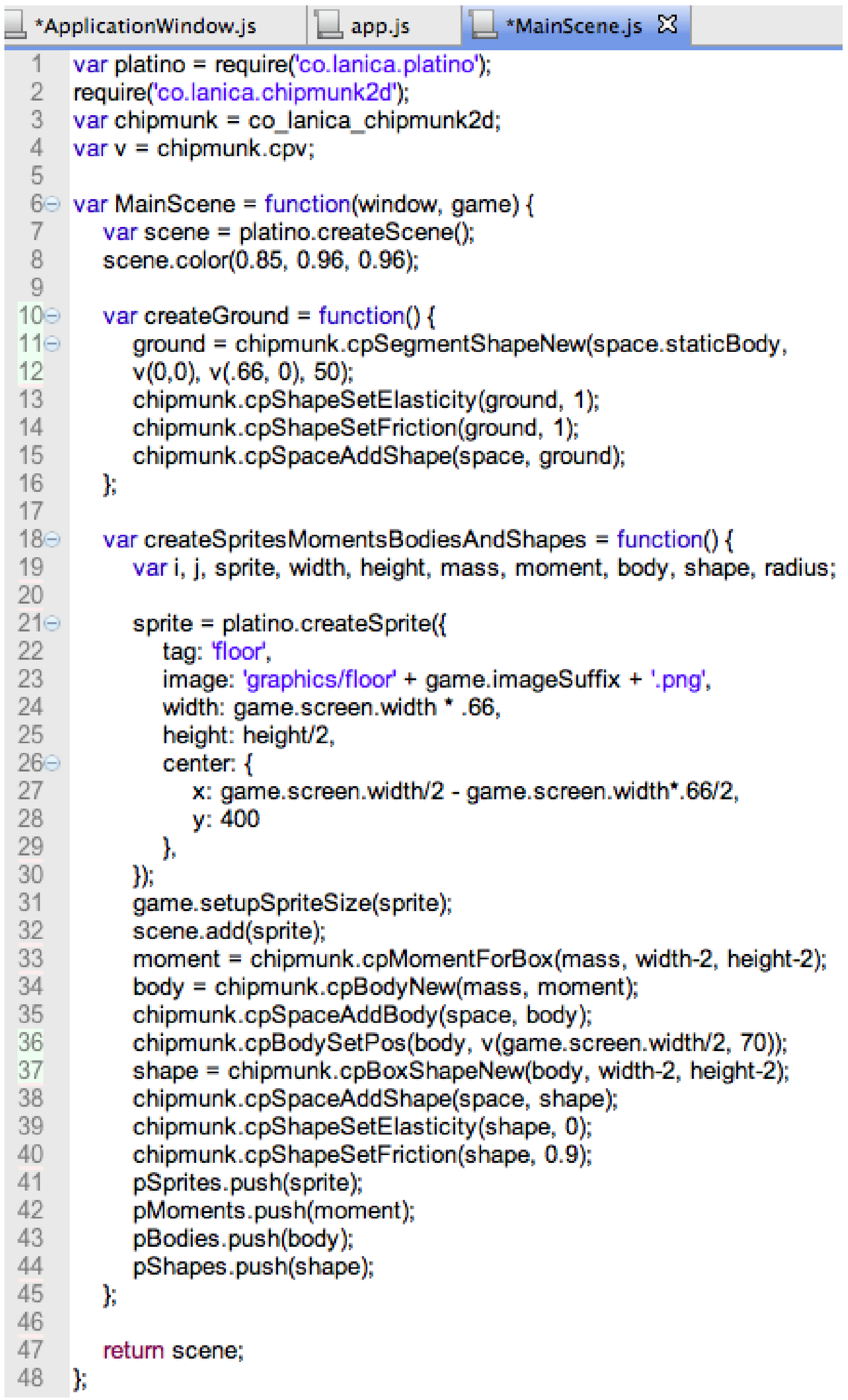


```
var platino = require('co.lanica.platino');
var ApplicationWindow = function() {
    var window = Ti.UI.createWindow({
        backgroundColor: 'black',
        fullscreen: true,
        navBarHidden: true
    });

    var game = platino.createGameView();
    game.fps = 30;
    game.color(0, 0, 0);
    var targetWidth = 320;
    var targetHeight = 480;

    game.addEventListener('onload', function(e) {
        updateScreenSize();
        var MainScene  = require("scenes/MainScene");
        game.currentScene = new MainScene(window, game);
        game.pushScene(game.currentScene);
        game.start();
    });

    game.setupSpriteSize = function(sprite) {
        var width = sprite.width / game.screenScale;
        var height = sprite.height / game.screenScale;
        sprite.width = (width < 1) ? 1 : width;
        sprite.height = (height < 1) ? 1 : height;
    };

    game.locationInView = function(_e) {
        var e = { type:_e.type, x:_e.x, y:_e.y, source:_e.source };
        var x = e.x * game.touchScaleX;
        var y = e.y * game.touchScaleY;
        e.x = x;
        e.y = y;
        return e;
    };

    window.add(game);
    return window;
```

Figure 50. Titanium Game App Code—UI.

Using the Platino game engine with the Titanium SDK requires that we use the Chipmunk2D physics engine, which is a "simulation layer that handles complex physics calculations…" [25]. Both Platino and Chipmunk 2D operate independently from one another so the developer must properly sync data from the two layers together so that sprites on screen can visible handle physics correctly.

After we require the Chipmunk2d module and create the game screen, the **locationInView** function must be created to convert screen touching coordinates to Platino coordinates. Other functions must be created by the developer to compensate for differences between Lanica’s Platino and the Chipmunk physics engine that it employs, which include screen coordinates differences (the y-values in each are of opposite values) (Figure 50) and angles (radians verses degrees) [25].

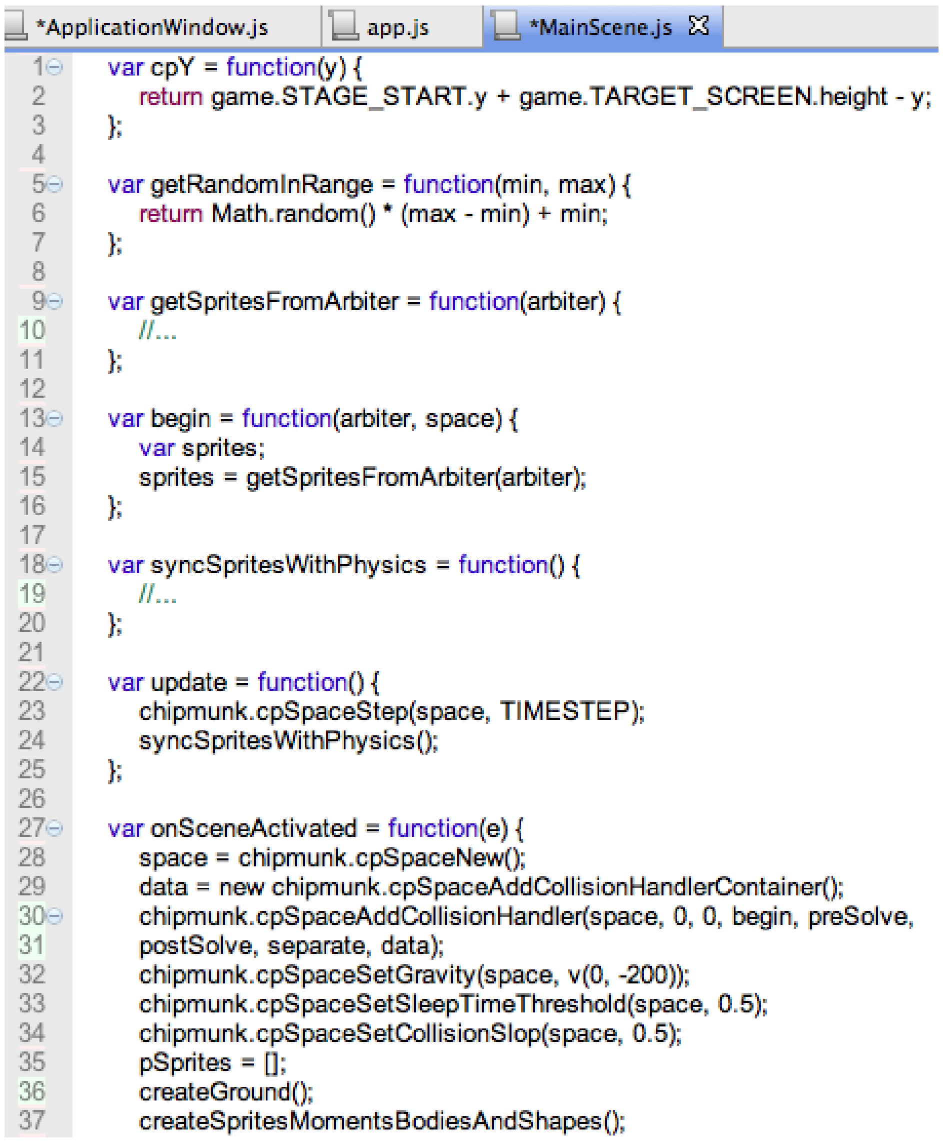


```
var platino = require('co.lanica.platino');
require('co.lanica.chipmunk2d');
var chipmunk = co_lanica_chipmunk2d;
var v = chipmunk.cpv;

var MainScene = function(window, game) {
    var scene = platino.createScene();
    scene.color(0.85, 0.96, 0.96);

    var createGround = function() {
        ground = chipmunk.cpSegmentShapeNew(space.staticBody,
        v(0,0), v(.66, 0), 50);
        chipmunk.cpShapeSetElasticity(ground, 1);
        chipmunk.cpShapeSetFriction(ground, 1);
        chipmunk.cpSpaceAddShape(space, ground);
    };

    var createSpritesMomentsBodiesAndShapes = function() {
        var i, j, sprite, width, height, mass, moment, body, shape, radius;

        sprite = platino.createSprite({
            tag: 'floor',
            image: 'graphics/floor' + game.imageSuffix + '.png',
            width: game.screen.width * .66,
            height: height/2,
            center: {
                x: game.screen.width/2 - game.screen.width*.66/2,
                y: 400
            },
        });
        game.setupSpriteSize(sprite);
        scene.add(sprite);
        moment = chipmunk.cpMomentForBox(mass, width-2, height-2);
        body = chipmunk.cpBodyNew(mass, moment);
        chipmunk.cpSpaceAddBody(space, body);
        chipmunk.cpBodySetPos(body, v(game.screen.width/2, 70));
        shape = chipmunk.cpBoxShapeNew(body, width-2, height-2);
        chipmunk.cpSpaceAddShape(space, shape);
        chipmunk.cpShapeSetElasticity(shape, 0);
        chipmunk.cpShapeSetFriction(shape, 0.9);
        pSprites.push(sprite);
        pMoments.push(moment);
        pBodies.push(body);
        pShapes.push(shape);
    };

    return scene;
};
```

Figure 51. Titanium Game App Code—UI cont.

Each of the UI objects used in this application is a sprite, which is a two-dimensional image used as a part of the game world. In Figure 51, we show how Platino (with the Chipmunk 2D module) allows these objects to react to physics by using structures in the **createSpitesMomentsBodiesAndShapes** function.

Event-handling is necessary for tapping the ball, having the ball respond to gravity, and collision events between the ball and floor. Figure 52 includes the code necessary for the application to handle these events. **Begin** is one of multiple functions used as a part of a collision handler container. Arbiters are the "structures that contain references to the two bodies that are involved in the collision…" [25].

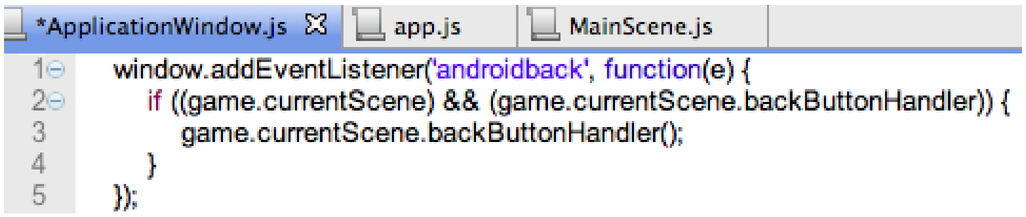


*ApplicationWindow.js   app.js   *MainScene.js

```
var cpY = function(y) {
    return game.STAGE_START.y + game.TARGET_SCREEN.height - y;
};

var getRandomInRange = function(min, max) {
    return Math.random() * (max - min) + min;
};

var getSpritesFromArbiter = function(arbiter) {
    //...
};

var begin = function(arbiter, space) {
    var sprites;
    sprites = getSpritesFromArbiter(arbiter);
};

var syncSpritesWithPhysics = function() {
    //...
};

var update = function() {
    chipmunk.cpSpaceStep(space, TIMESTEP);
    syncSpritesWithPhysics();
};

var onSceneActivated = function(e) {
    space = chipmunk.cpSpaceNew();
    data = new chipmunk.cpSpaceAddCollisionHandlerContainer();
    chipmunk.cpSpaceAddCollisionHandler(space, 0, 0, begin, preSolve,
    postSolve, separate, data);
    chipmunk.cpSpaceSetGravity(space, v(0, -200));
    chipmunk.cpSpaceSetSleepTimeThreshold(space, 0.5);
    chipmunk.cpSpaceSetCollisionSlop(space, 0.5);
    pSprites = [];
    createGround();
    createSpritesMomentsBodiesAndShapes();
```

Figure 52. Titanium Game App Code—Program Control.

Figure 53 shows the event listener for the Android backbutton, a simple Android-specific JavaScript that does not have to be removed or modified when building for the iOS.



```
window.addEventListener('androidback', function(e) {
    if ((game.currentScene) && (game.currentScene.backButtonHandler)) {
        game.currentScene.backButtonHandler();
    }
});
```

Figure 53.   Titanium Game App Code—Program Control cont.

3. Device-Accessing Application

Figures 54 and 55 show the screens of the finished product on Android and iOS devices. Note how Titanium uses widgets based on the platform the application is deployed on; one codebase outputs native buttons and platforms and maps (iOS uses Apple Maps and Android uses Google Maps).

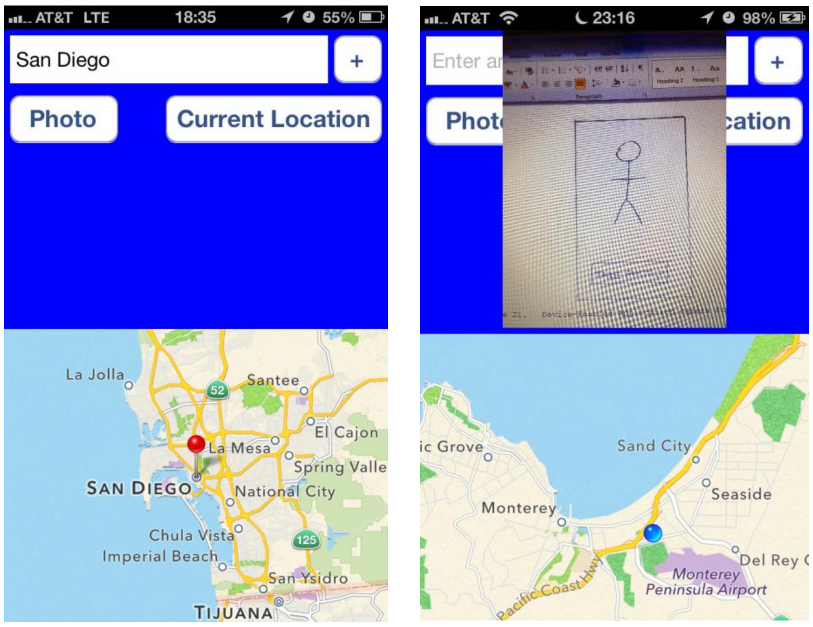


Figure 54.   iOS Device-Accessing Application.

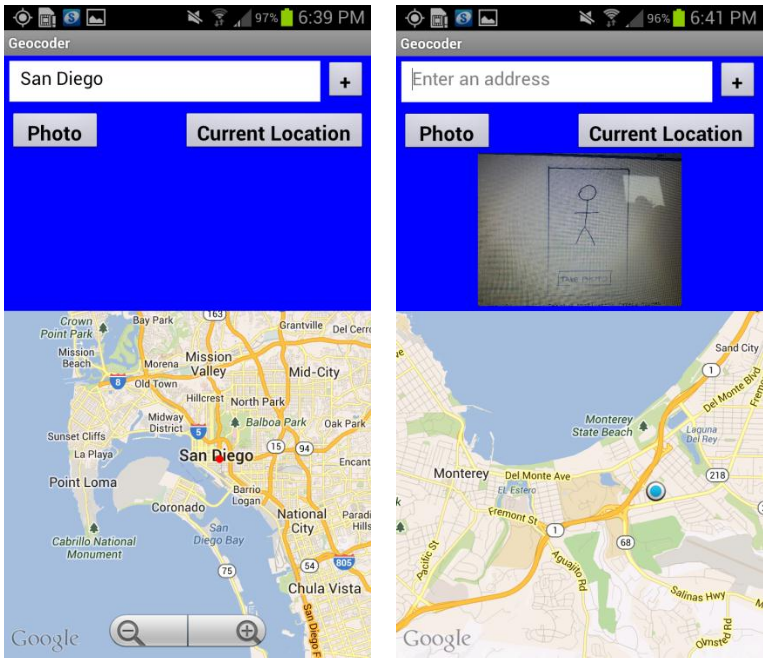
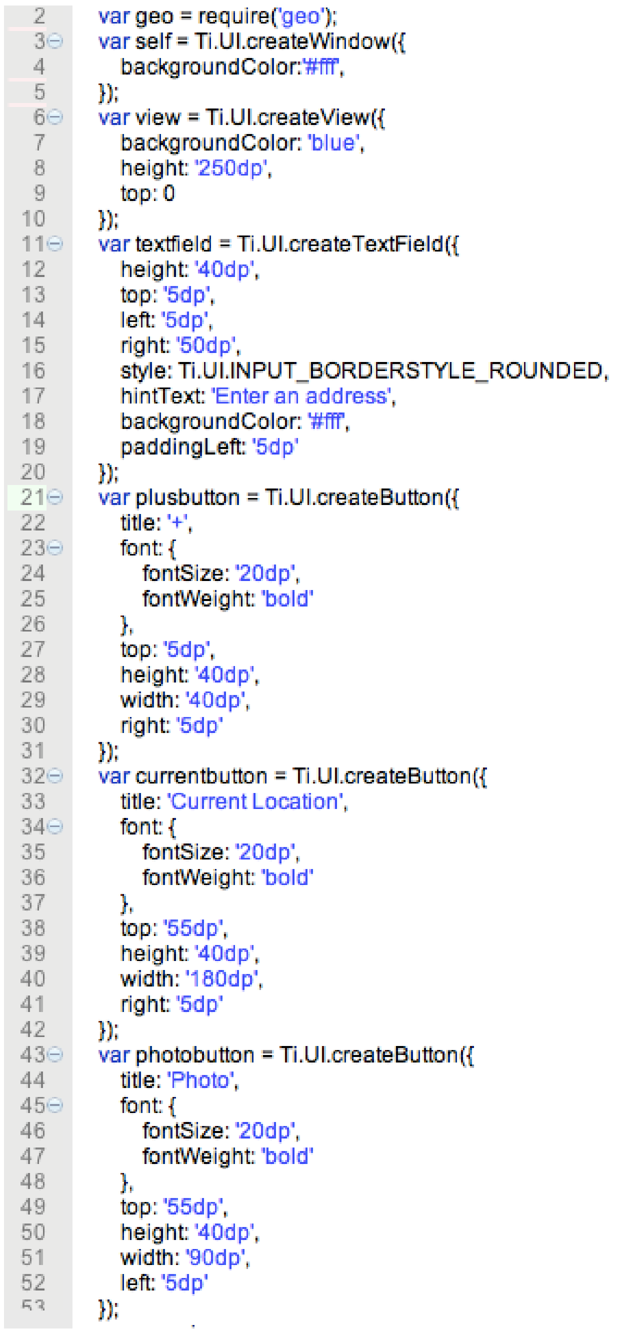


Figure 55. Titanium Android Device-Accessing Application.

The user interface (including the buttons, textbox, and map) and program control (including event-handlers and access to device features) are implemented in Figures 56 through 58.

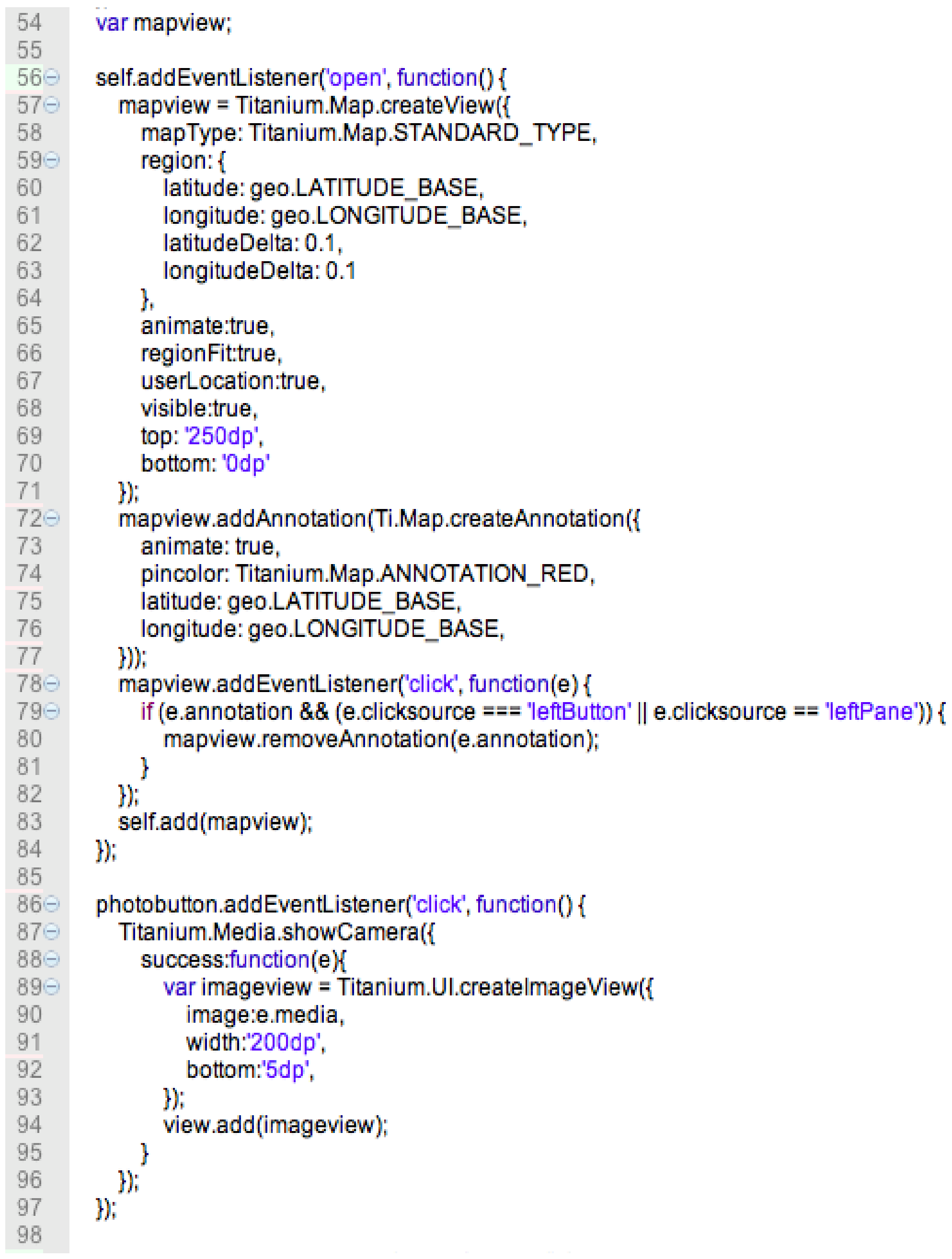
In Figure 56, we first require the geo.js module so that we can later access the **forwardGeocoder** and **reverseGeocoder** methods to convert between addresses and geographic coordinates [26]. UI objects are created, positioned, later added to the View container, and then added along with the View to the window.



```
var geo = require('geo');
var self = Ti.UI.createWindow({
   backgroundColor:'#fff',
});
var view = Ti.UI.createView({
   backgroundColor: 'blue',
   height: '250dp',
   top: 0
});
var textfield = Ti.UI.createTextField({
   height: '40dp',
   top: '5dp',
   left: '5dp',
   right: '50dp',
   style: Ti.UI.INPUT_BORDERSTYLE_ROUNDED,
   hintText: 'Enter an address',
   backgroundColor: '#fff',
   paddingLeft: '5dp'
});
var plusbutton = Ti.UI.createButton({
   title: '+',
   font: {
      fontSize: '20dp',
      fontWeight: 'bold'
   },
   top: '5dp',
   height: '40dp',
   width: '40dp',
   right: '5dp'
});
var currentbutton = Ti.UI.createButton({
   title: 'Current Location',
   font: {
      fontSize: '20dp',
      fontWeight: 'bold'
   },
   top: '55dp',
   height: '40dp',
   width: '180dp',
   right: '5dp'
});
var photobutton = Ti.UI.createButton({
   title: 'Photo',
   font: {
      fontSize: '20dp',
      fontWeight: 'bold'
   },
   top: '55dp',
   height: '40dp',
   width: '90dp',
   left: '5dp'
});
```

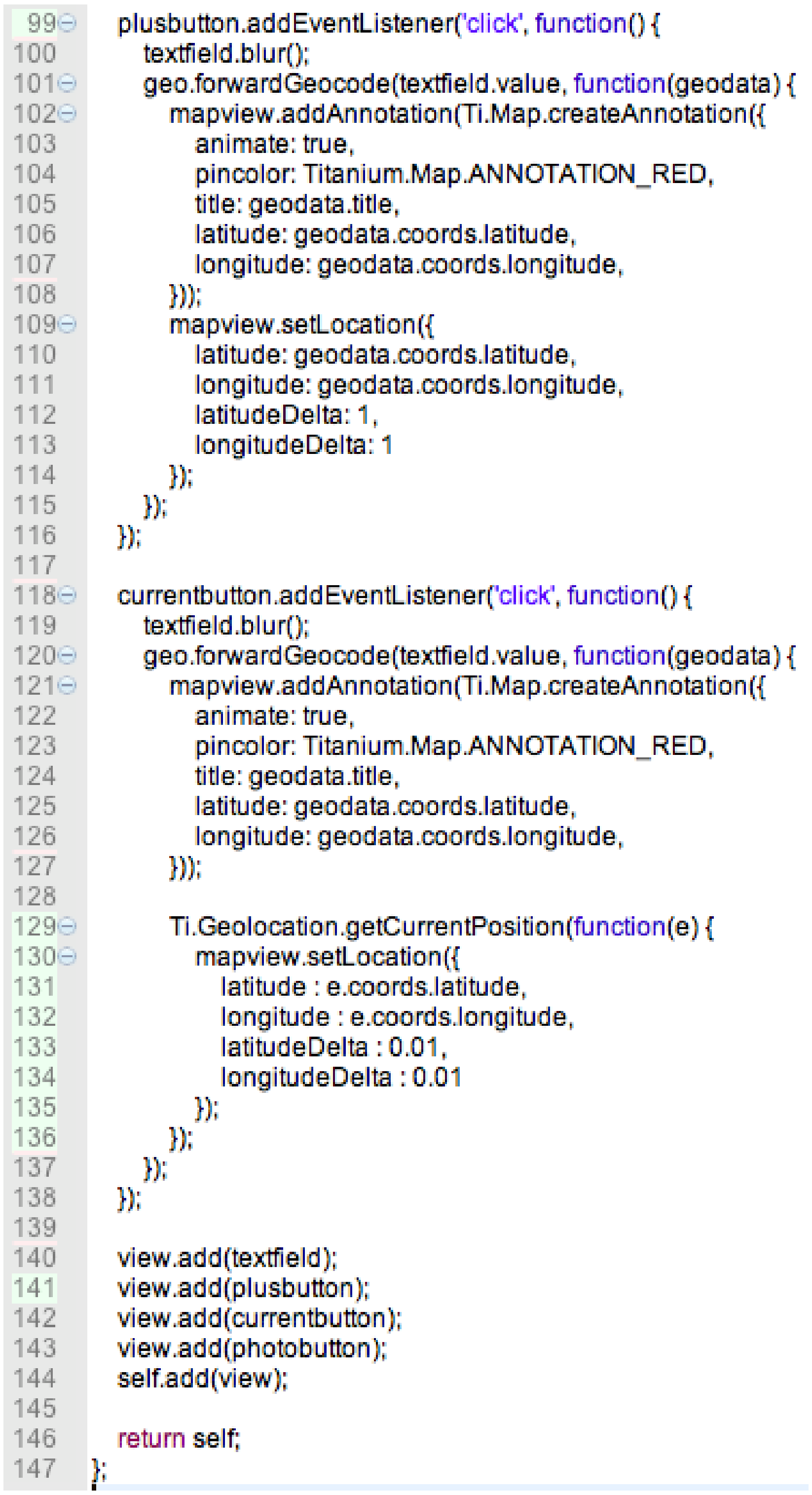
Figure 56. Titanium Device-Accessing App Code.

Figure 57 and 58 show our event-handlers. Upon opening, we create a MapView and add it to the screen. The plusButton click-event listener uses forward geocoding to locate the textfield inputted address and display the location on the map using coordinates. For the photobutton listener, we utilize Titanium’s Media.showCamera function show the camera and then, upon successfully receiving a photo from either the camera or the photo library, an imageView is created with the photo and then added to the view.



```
var mapview;

self.addEventListener('open', function() {
    mapview = Titanium.Map.createView({
        mapType: Titanium.Map.STANDARD_TYPE,
        region: {
            latitude: geo.LATITUDE_BASE,
            longitude: geo.LONGITUDE_BASE,
            latitudeDelta: 0.1,
            longitudeDelta: 0.1
        },
        animate:true,
        regionFit:true,
        userLocation:true,
        visible:true,
        top: '250dp',
        bottom: '0dp'
    });
    mapview.addAnnotation(Ti.Map.createAnnotation({
        animate: true,
        pincolor: Titanium.Map.ANNOTATION_RED,
        latitude: geo.LATITUDE_BASE,
        longitude: geo.LONGITUDE_BASE,
    }));
    mapview.addEventListener('click', function(e) {
        if (e.annotation && (e.clicksource === 'leftButton' || e.clicksource == 'leftPane')) {
            mapview.removeAnnotation(e.annotation);
        }
    });
    self.add(mapview);
});

photobutton.addEventListener('click', function() {
    Titanium.Media.showCamera({
        success:function(e){
            var imageview = Titanium.UI.createImageView({
                image:e.media,
                width:'200dp',
                bottom:'5dp',
            });
            view.add(imageview);
        }
    });
});
```

Figure 57. Titanium Device-Accessing App Code.



```
    plusbutton.addEventListener('click', function() {
        textfield.blur();
        geo.forwardGeocode(textfield.value, function(geodata) {
            mapview.addAnnotation(Ti.Map.createAnnotation({
                animate: true,
                pincolor: Titanium.Map.ANNOTATION_RED,
                title: geodata.title,
                latitude: geodata.coords.latitude,
                longitude: geodata.coords.longitude,
            }));
            mapview.setLocation({
                latitude: geodata.coords.latitude,
                longitude: geodata.coords.longitude,
                latitudeDelta: 1,
                longitudeDelta: 1
            });
        });
    });

    currentbutton.addEventListener('click', function() {
        textfield.blur();
        geo.forwardGeocode(textfield.value, function(geodata) {
            mapview.addAnnotation(Ti.Map.createAnnotation({
                animate: true,
                pincolor: Titanium.Map.ANNOTATION_RED,
                title: geodata.title,
                latitude: geodata.coords.latitude,
                longitude: geodata.coords.longitude,
            }));

            Ti.Geolocation.getCurrentPosition(function(e) {
                mapview.setLocation({
                    latitude : e.coords.latitude,
                    longitude : e.coords.longitude,
                    latitudeDelta : 0.01,
                    longitudeDelta : 0.01
                });
            });
        });
    });

    view.add(textfield);
    view.add(plusbutton);
    view.add(currentbutton);
    view.add(photobutton);
    self.add(view);

    return self;
};
```

Figure 58. Titanium Device-Accessing App Code.

C. PHONEGAP

1. Productivity Application

Figures 59 and 60 show the screens of the finished product on both Android and iOS devices. Since it is a web application embedded in a native wrapper, it looks visually the same on each device. Appery.io (utilizing PhoneGap) is the development tool that we used.

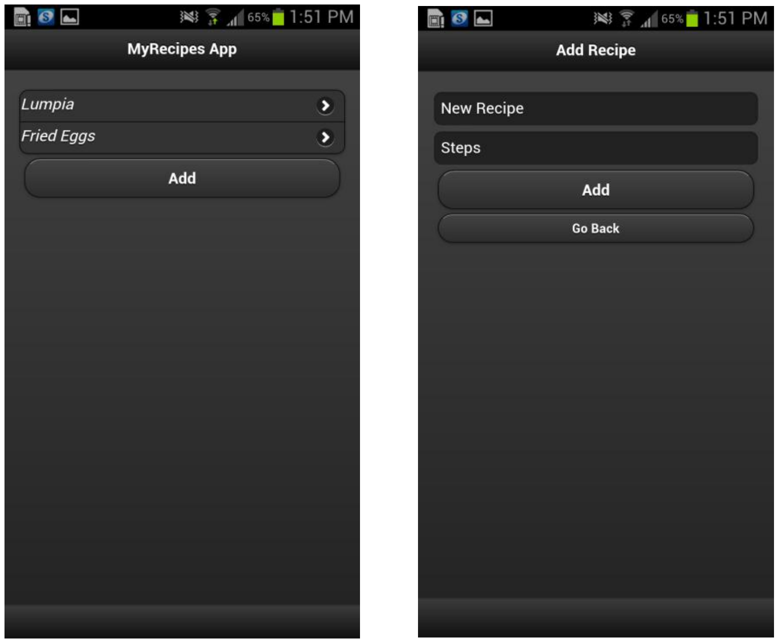
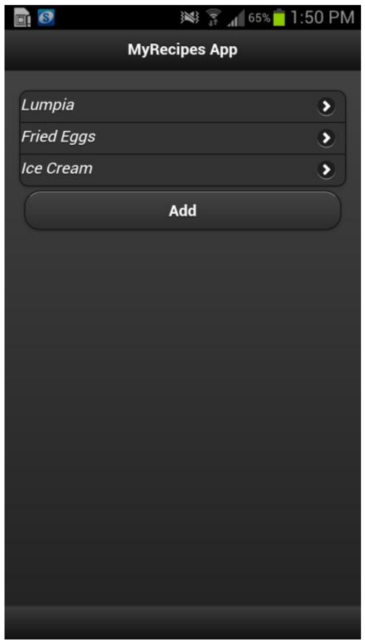


Figure 59. PhoneGap Productivity Application.



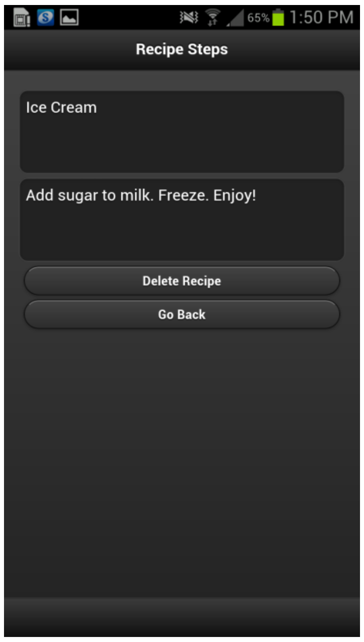
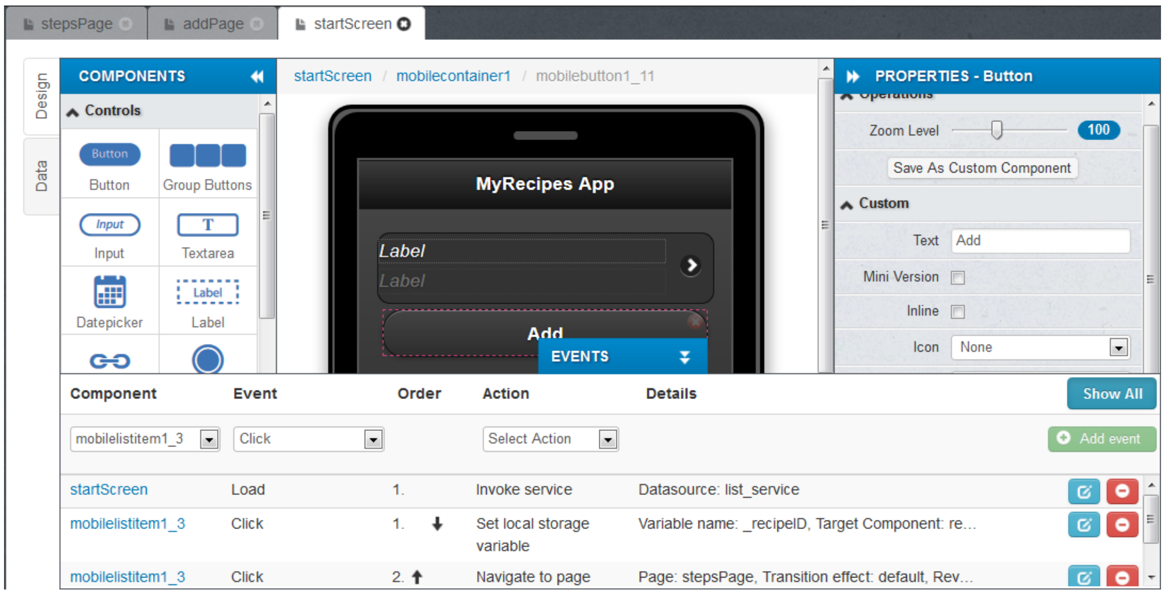


Figure 60. PhoneGap Productivity Application cont.

In the main window, there is a list view and two buttons. Figure 61 displays how we construct this window. Notice the drag-and-drop UI components as well as the properties sidebar, which provide for codeless development. Appery.io’s visual editor is used extensively for each of the three metric components.



Zoom Level 100

Text: Add

Mini Version ☐

Inline ☐

Icon: None

Component	Event	Order	Action	Details
mobilelistitem1_3	Click		Select Action	
startScreen	Load	1.	Invoke service	Datasource: list_service
mobilelistitem1_3	Click	1.	Set local storage variable	Variable name: _recipeID, Target Component: re...
mobilelistitem1_3	Click	2.	Navigate to page	Page: stepsPage, Transition effect: default, Rev...

Figure 61. Appery.io Productivity Start Screen Development.

When an item in the list is selected (by the end-user tapping on it), the second window is displayed to show the details of the selected item. This transition from one window to another is achieved by the **set local variable** and **navigate to page** actions.

The event-handler bar displays the events we've selected. To invoke the service that adds all database items to the UI list, we add it to the page and then edit its response mapping as shown in Figure 62. The name **$** is the array of all recipes, and we map each recipe to the UI list. **Recipe** is the name of the specific recipe. **_id** is the key used later by the steps page to retrieve the details of the specific recipe.

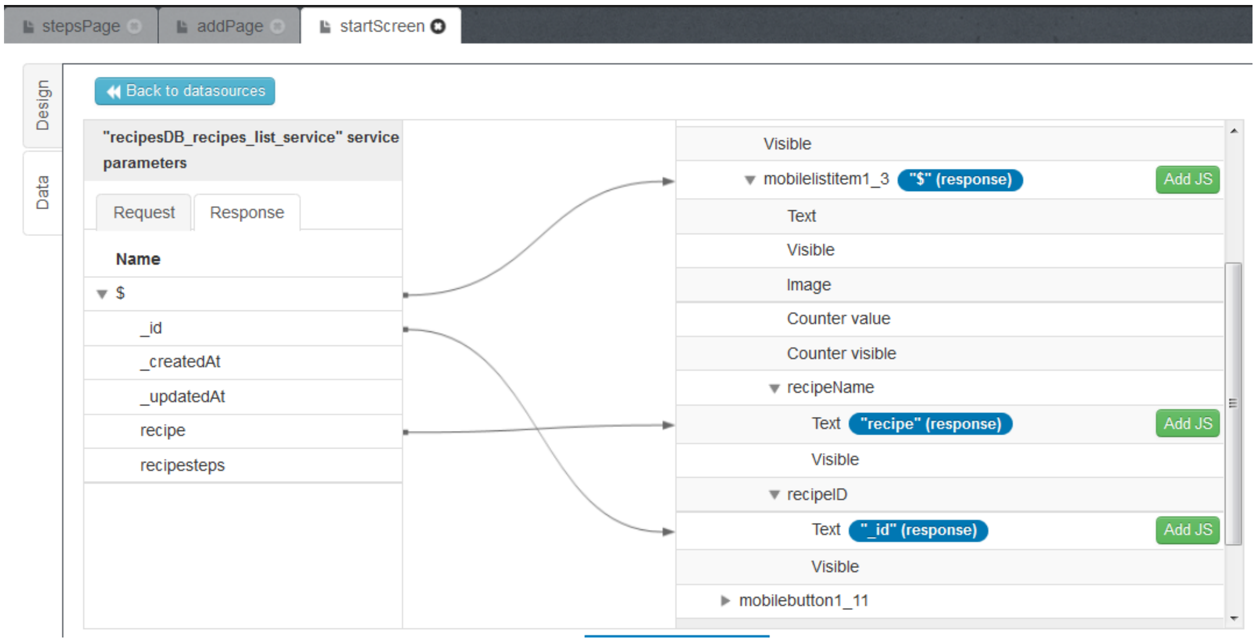
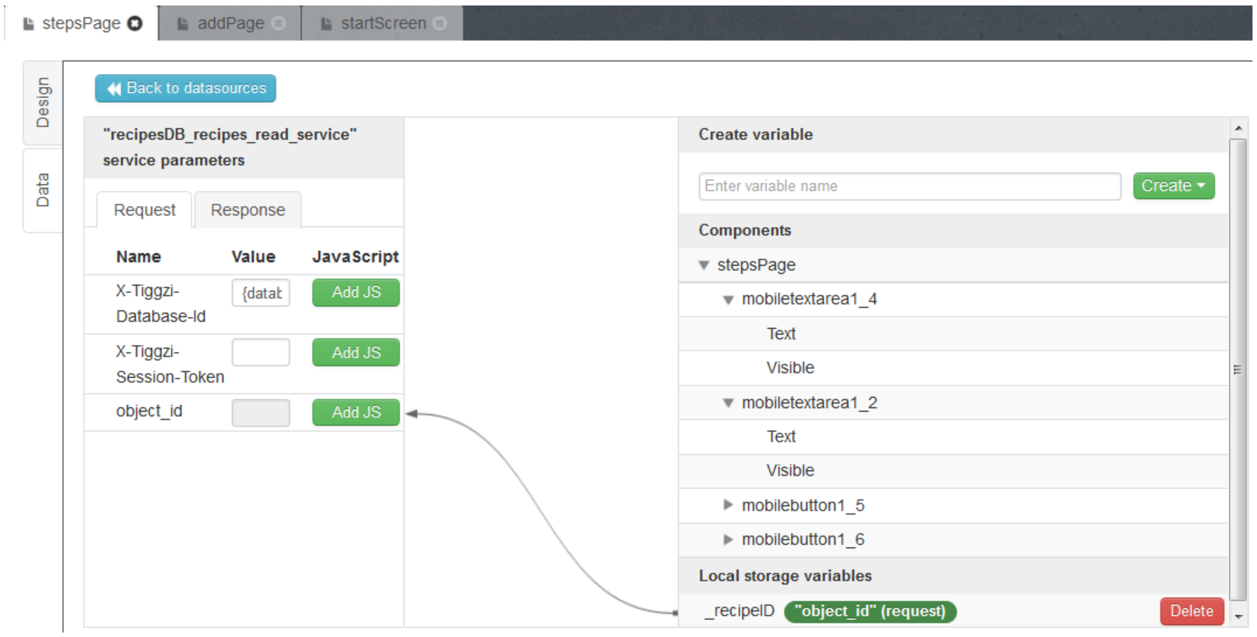


Figure 62. Appery.io Productivity Start Screen Data Response.



Name	Value	JavaScript
X-Tiggzi-Database-Id	{datat	Add JS
X-Tiggzi-Session-Token		Add JS
object_id		Add JS

Figure 63. Appery.io Productivity Steps Page Data Request.

In Figure 63, we get the recipe's ID from local storage and map it as the object ID. In Figure 64 the recipe name is mapped to the upper textbox and the recipe steps map to the lower textbox. Both the recipe name and steps have the ID as the key.

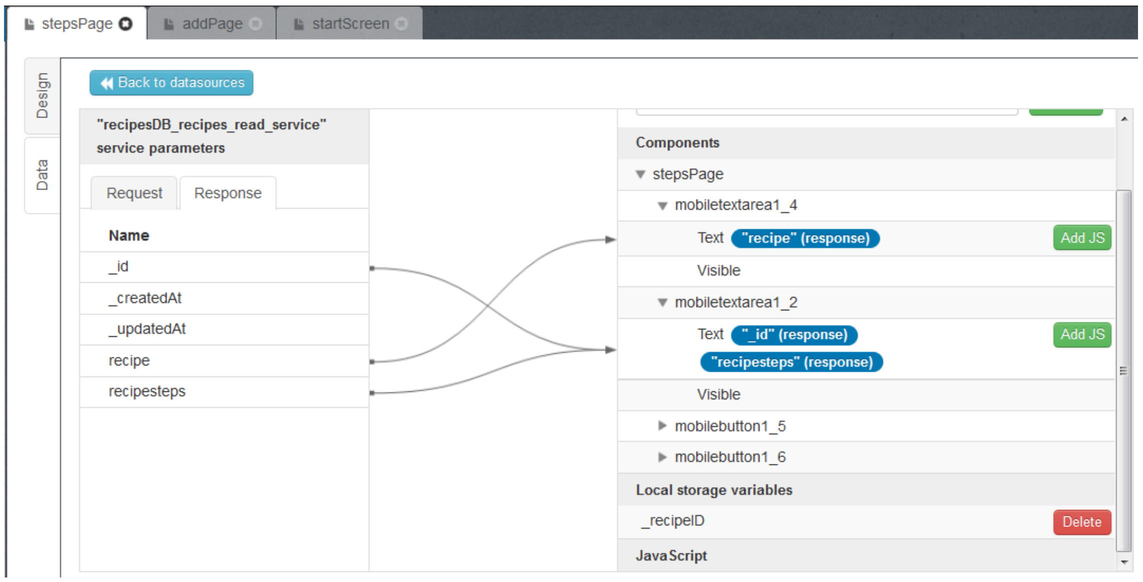
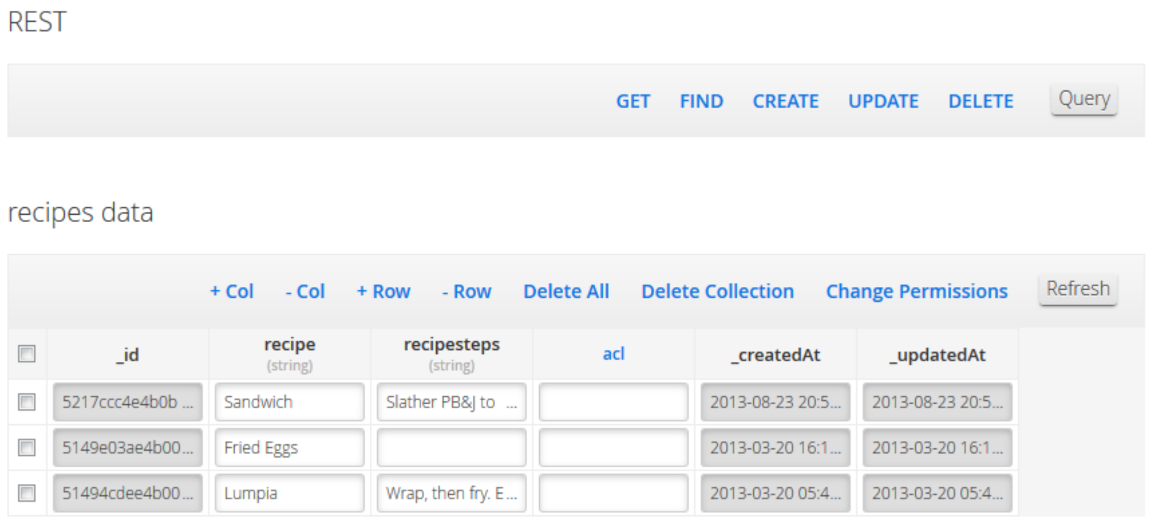


Figure 64. Appery.io Productivity Steps Page Data Response.

The recipe data are managed by Appery.io's cloud-based relational database. An advantage of this is that the only time local storage is required is when saving a recipe ID so that it can be deleted on command. Another advantage is that Appery.io databases can import and utilize REST (representational state transfer) APIs to conduct many services, including loading, adding, and deleting [27]. Local storage for this application's database is possible by utilizing SQLite3, that requiring coding on top of visual editing [27].

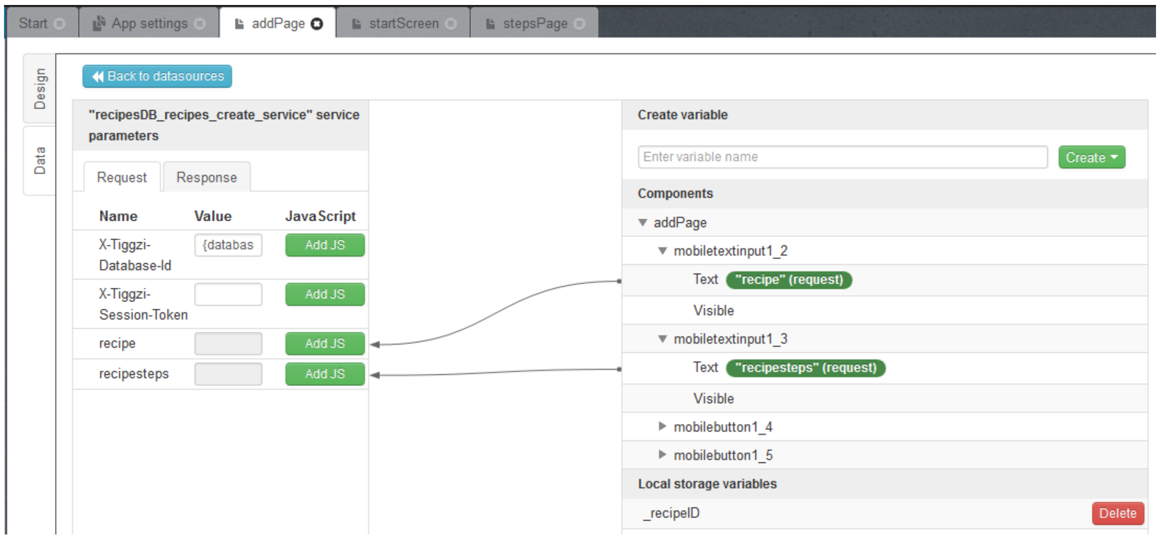
Database table creation is done by simply adding a recipe and steps column to the recipes data collection and filling in any entries (Figure 65).



_id	recipe (string)	recipesteps (string)	acl	_createdAt	_updatedAt
5217ccc4e4b0b ...	Sandwich	Slather PB&J to ...		2013-08-23 20:5...	2013-08-23 20:5...
5149e03ae4b00...	Fried Eggs			2013-03-20 16:1...	2013-03-20 16:1...
51494cdee4b00...	Lumpia	Wrap, then fry. E...		2013-03-20 05:4...	2013-03-20 05:4...

Figure 65. Appery.io Productivity Database Create.

The end user can add a new recipe and also delete any existing recipe by tapping the corresponding add and delete buttons. The tap events for the two buttons are implemented in Figure 66 and 67, respectively.



"recipesDB_recipes_create_service" service parameters

Request | Response

Name	Value	JavaScript
X-Tiggzi-Database-Id	{databas	Add JS
X-Tiggzi-Session-Token		Add JS
recipe		Add JS
recipesteps		Add JS

Create variable

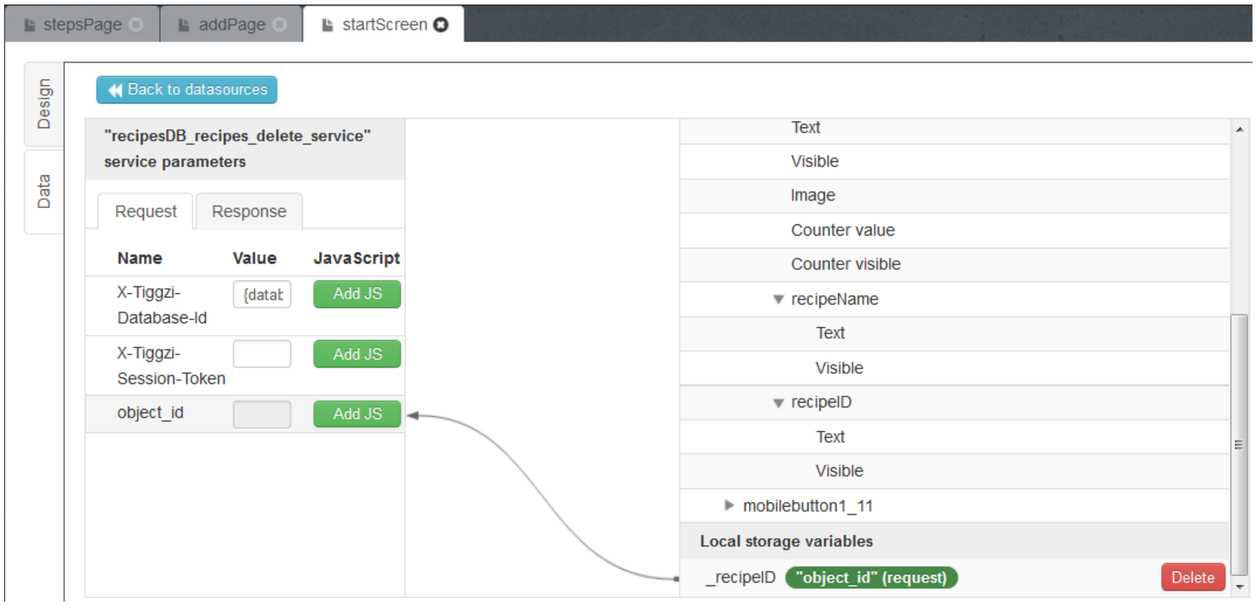
Enter variable name

Components
- addPage
  - mobiletextinput1_2
    - Text "recipe" (request)
    - Visible
  - mobiletextinput1_3
    - Text "recipesteps" (request)
    - Visible
  - mobilebutton1_4
  - mobilebutton1_5

Local storage variables
- _recipeID

Figure 66. Appery.io Productivity Add Page Data Request.

The create service maps the inputs from the text boxes to the recipe and recipe steps. The delete service only requires the mapping of the recipe ID (obtained from local storage again) to the object ID.



"recipesDB_recipes_delete_service" service parameters

Request | Response

Name	Value	JavaScript
X-Tiggzi-Database-Id	{datat	Add JS
X-Tiggzi-Session-Token		Add JS
object_id		Add JS

- Text
- Visible
- Image
- Counter value
- Counter visible
- recipeName
  - Text
  - Visible
- recipeID
  - Text
  - Visible
- mobilebutton1_11

Local storage variables
- _recipeID "object_id" (request)

Figure 67. Appery.io Productivity Start Screen Data Request.

2. Game Application

As September, 2013, Appery.io does not support a tool for graphics-based games. Its niche in the cross-platform application development world is to be a "cloud-based platform with visual development tools and integrated backend services" [3], mainly benefitting those who intend on building enterprise applications [3].

PhoneGap itself can be used though. A game can be developed as if intended for use in a browser, but instead, wrapped in a native platform wrapper by using PhoneGap. Figure 68 is a screenshot of our PhoneGap game application, where we used HTML5 to create a web application and then PhoneGap Build for the company server to build the platform-specific version of our application. Due to the limited time we had to work with PhoneGap Build, we did not include code for either a home screen or an option screen.

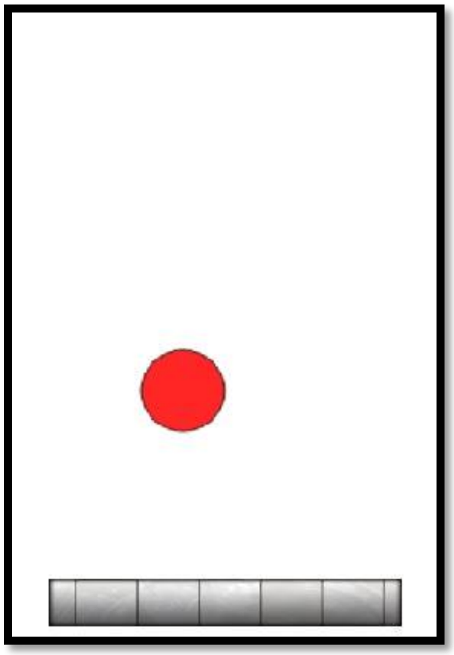
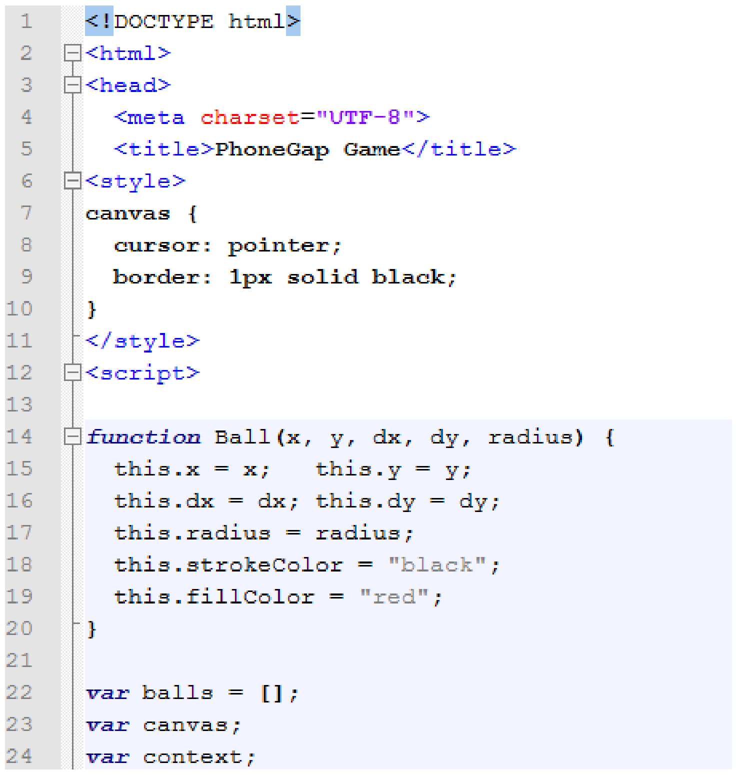


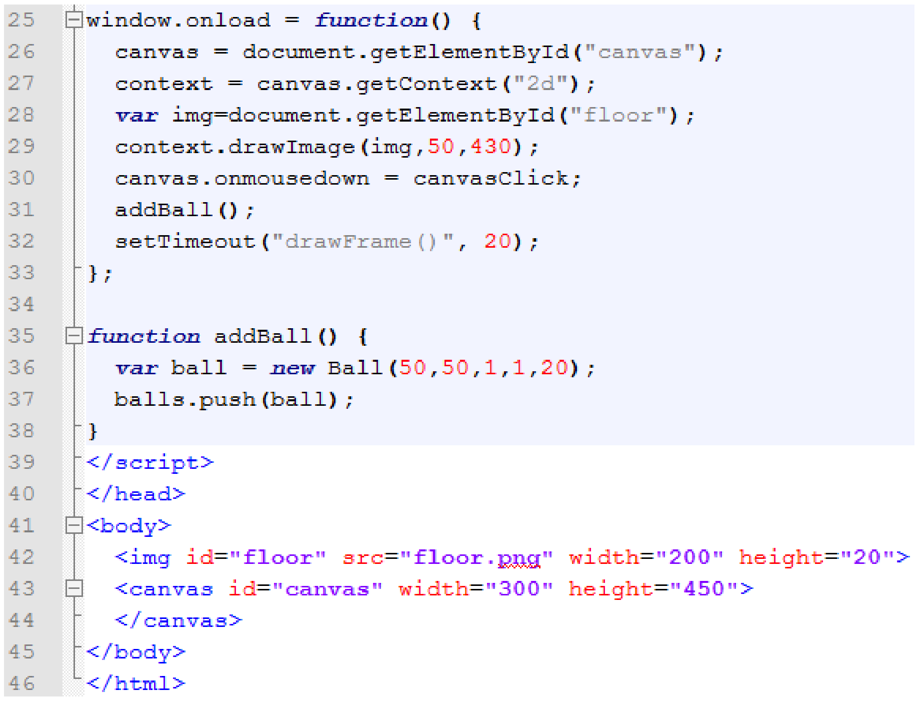
Figure 68. PhoneGap Game Application

We used HTML5 and JavaScript to create the web application. Construction of the UI objects and event-handlers was accomplished as follows (Figures 69 through 71):



```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>PhoneGap Game</title>
<style>
canvas {
  cursor: pointer;
  border: 1px solid black;
}
</style>
<script>

function Ball(x, y, dx, dy, radius) {
  this.x = x;   this.y = y;
  this.dx = dx; this.dy = dy;
  this.radius = radius;
  this.strokeColor = "black";
  this.fillColor = "red";
}

var balls = [];
var canvas;
var context;
```

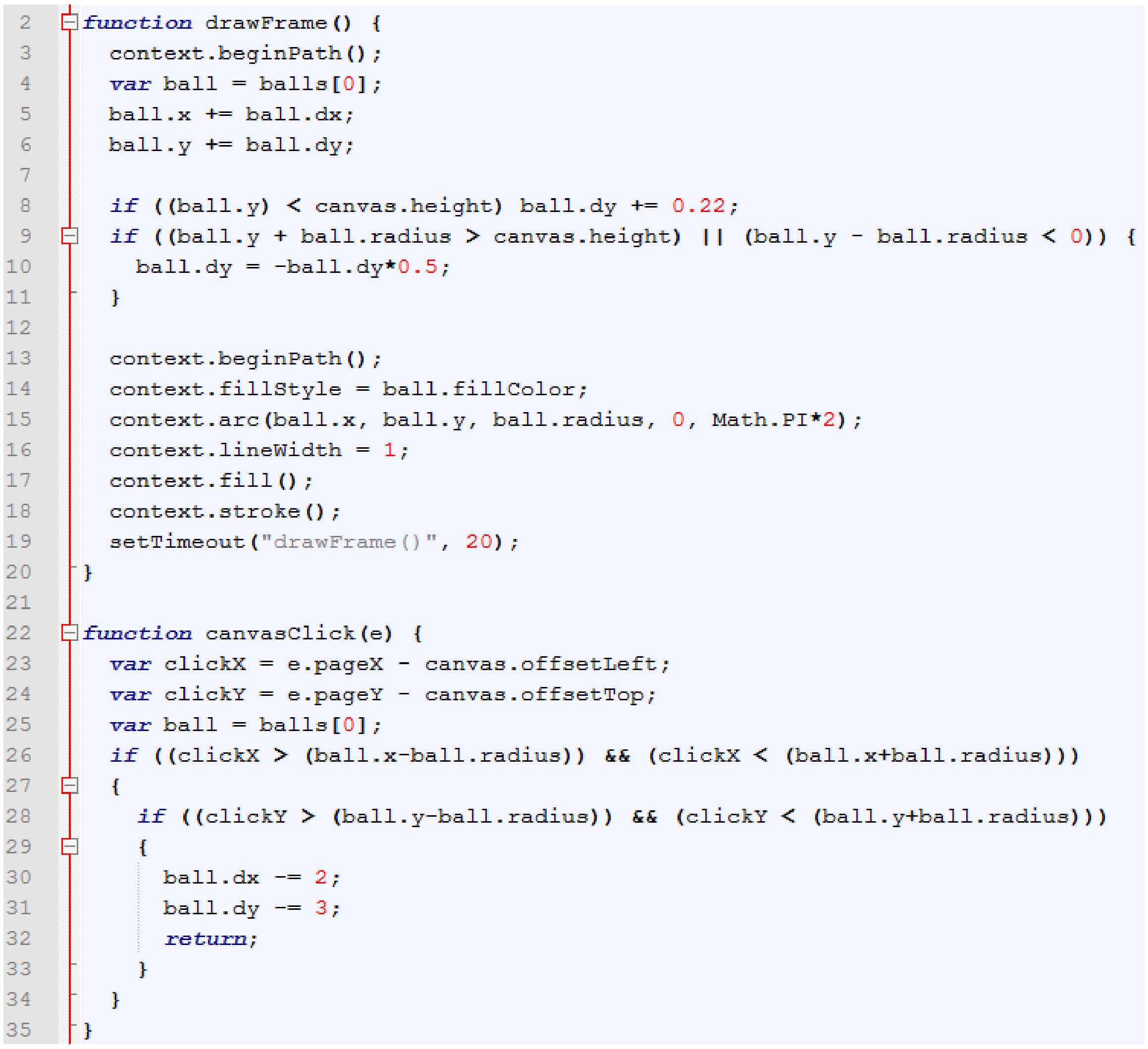
Figure 69. PhoneGap Game—UI Code.



```
window.onload = function() {
  canvas = document.getElementById("canvas");
  context = canvas.getContext("2d");
  var img=document.getElementById("floor");
  context.drawImage(img,50,430);
  canvas.onmousedown = canvasClick;
  addBall();
  setTimeout("drawFrame()", 20);
};

function addBall() {
  var ball = new Ball(50,50,1,1,20);
  balls.push(ball);
}
</script>
</head>
<body>
  <img id="floor" src="floor.png" width="200" height="20">
  <canvas id="canvas" width="300" height="450">
  </canvas>
</body>
</html>
```

Figure 70. PhoneGap Game—UI Code cont.

Figures 69 and 70 show how a function is used to create a Ball object with size, position, and speed attributes. After a canvas the size of a mobile screen is drawn, the **addBall()** function creates a new ball object and adds it on screen. The floor ID from the HTML body is used to include the floor image in the canvas.



```
function drawFrame() {
  context.beginPath();
  var ball = balls[0];
  ball.x += ball.dx;
  ball.y += ball.dy;

  if ((ball.y) < canvas.height) ball.dy += 0.22;
  if ((ball.y + ball.radius > canvas.height) || (ball.y - ball.radius < 0)) {
    ball.dy = -ball.dy*0.5;
  }

  context.beginPath();
  context.fillStyle = ball.fillColor;
  context.arc(ball.x, ball.y, ball.radius, 0, Math.PI*2);
  context.lineWidth = 1;
  context.fill();
  context.stroke();
  setTimeout("drawFrame()", 20);
}

function canvasClick(e) {
  var clickX = e.pageX - canvas.offsetLeft;
  var clickY = e.pageY - canvas.offsetTop;
  var ball = balls[0];
  if ((clickX > (ball.x-ball.radius)) && (clickX < (ball.x+ball.radius)))
  {
    if ((clickY > (ball.y-ball.radius)) && (clickY < (ball.y+ball.radius)))
    {
      ball.dx -= 2;
      ball.dy -= 3;
      return;
    }
  }
}
```

Figure 71. PhoneGap Game—Program Control Code.

While Figure 71 includes some UI related code, the intention of this figure is to show how physics and event handling is applied. Lines 8-10 are the conditionals that include formulas for simulating gravity and bouncing upon collision. **canvasClick()** is our touch event-handler that changes the Ball object's speed attributes based on if the canvas coordinates of the touch event matches the canvas coordinates of the ball.

3. Device-Accessing Application

As with the productivity application, we use Appery.io with PhoneGap to develop our device-accessing application. Figure 72 shows the screens of the finished product, which looks visually the same on both Android and iOS devices.

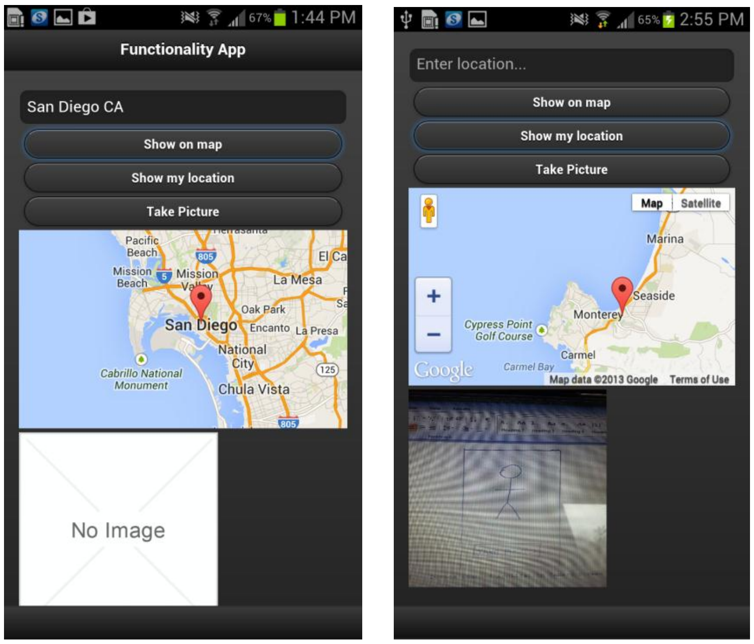
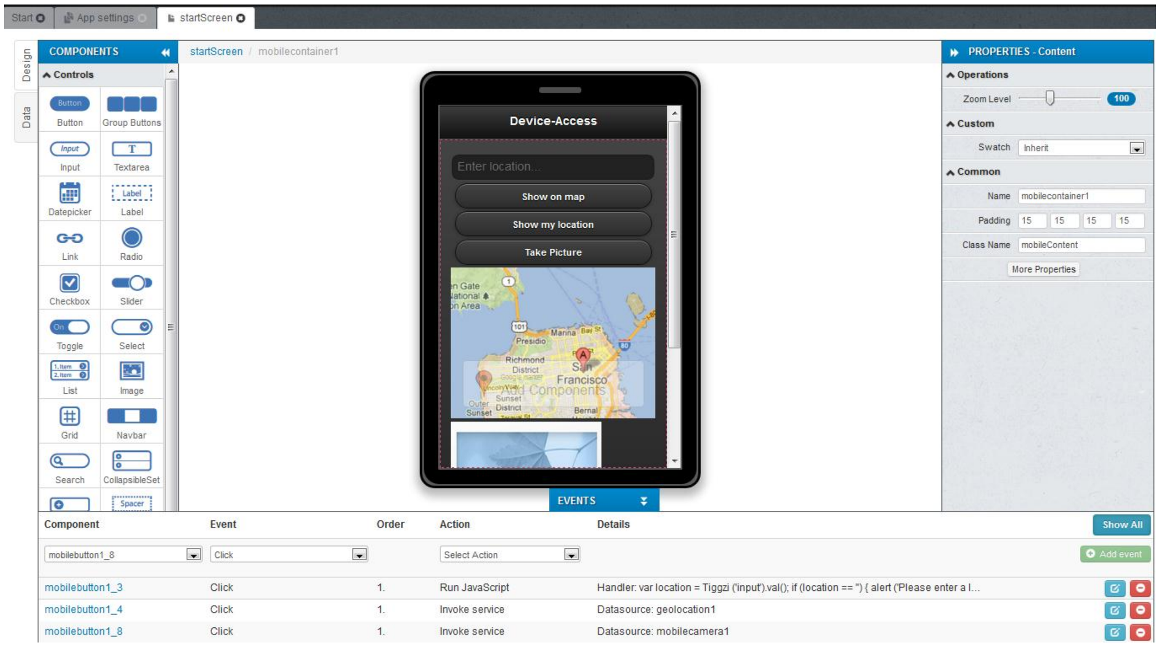


Figure 72. Appery.io Device-Accessing Application.

The user interface, including the Google Map component and the Image component, is constructed using the visual editor like in the productivity application (Figure 73).

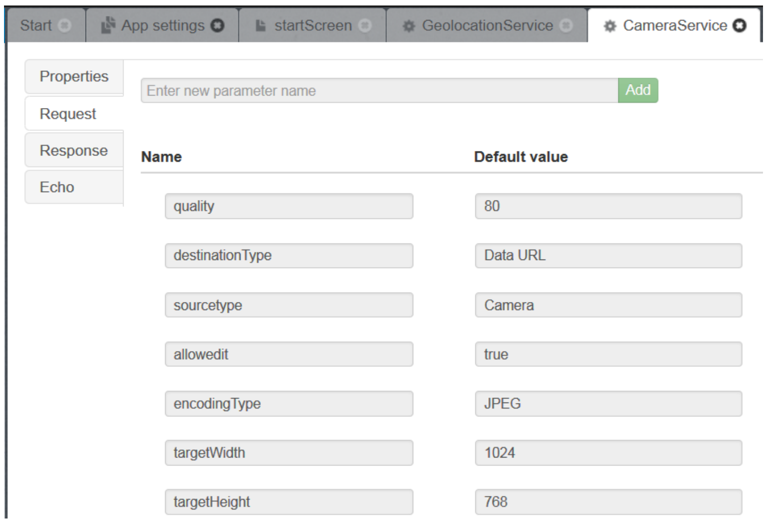


Swatch: Inherit
Name: mobilecontainer1
Padding: 15 15 15 15
Class Name: mobileContent

Component	Event	Order	Action	Details
mobilebutton1_8	Click		Select Action	
mobilebutton1_3	Click	1.	Run JavaScript	Handler: var location = Tiggzi ('input').val(); if (location == '') { alert ('Please enter a l...
mobilebutton1_4	Click	1.	Invoke service	Datasource: geolocation1
mobilebutton1_8	Click	1.	Invoke service	Datasource: mobilecamera1

Figure 73. Appery.io Device-Accessing Start Screen Development

Both the camera and geolocation services, which are provided by the PhoneGap bridge, are added to the application. Both services are pre-configured, and Figure 74 shows the configuration of the camera service.



Name	Default value
quality	80
destinationType	Data URL
sourcetype	Camera
allowedit	true
encodingType	JPEG
targetWidth	1024
targetHeight	768

Figure 74. Appery.io Device-Accessing Service Configuration

Figures 75 and 76 detail two of our event-handlers: how we map our data from the service to the respective UI component after a button has been tapped.

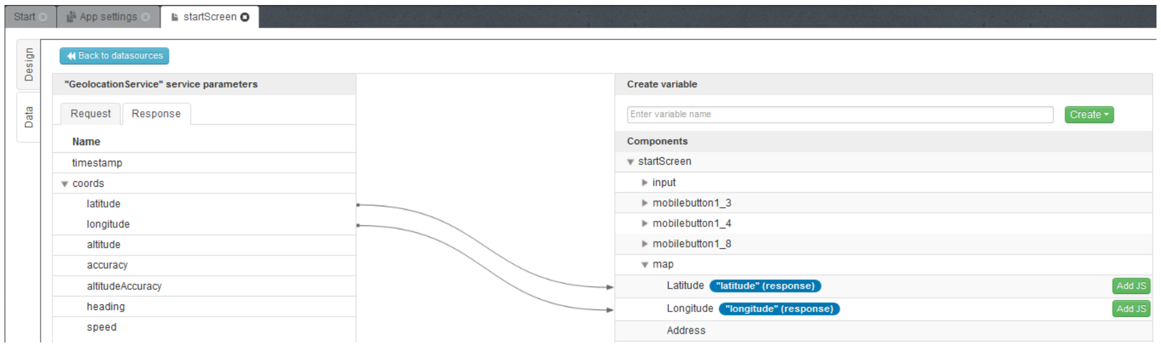


Figure 75. Appery.io Device Accessing GeoLocation Response.

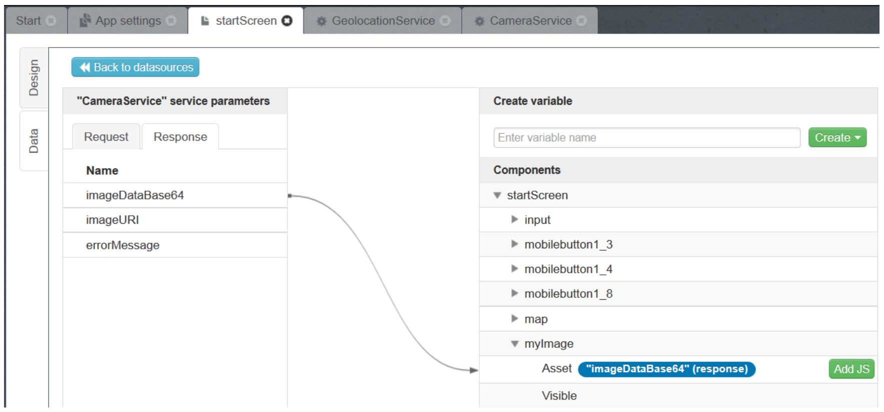
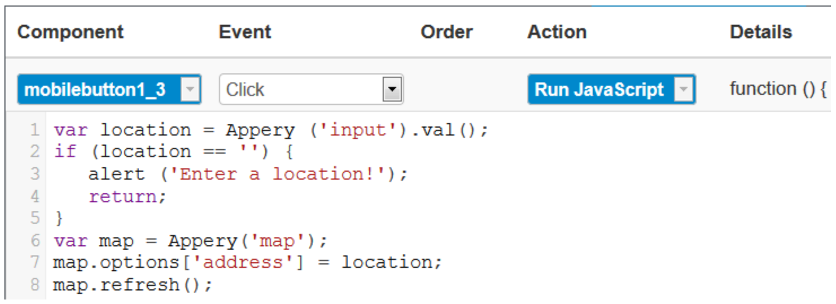


Figure 76. Appery.io Device-Accessing Camera Response.

In Figure 77, we demonstrate how JavaScript can functions can be edited, like how the platform-specific source code and web application PhoneGap code can be.



```
function () {
var location = Appery ('input').val();
if (location == '') {
   alert ('Enter a location!');
   return;
}
var map = Appery('map');
map.options['address'] = location;
map.refresh();
```

Figure 77. Appery.io Device-Accessing Location Finding.

THIS PAGE INTENTIONALLY LEFT BLANK

V. FINAL ANALYSIS

In this chapter we conduct our comparative analysis on our three mobile cross-platform development tools. This includes discussion on the challenges and approaches that we took. As described in Chapter 3, we analyze three cross-platform development tools in three areas: user interface specification, program control, and database management. We include tables ranking the relative productivity we had when using each development tool on each application by component, and with a rank of "1" as the most productive of the three. By conducting our research through the development of applications, we have learned more differences as well as similarities between the productiveness that developers can achieve when using each of three tools.

A. USER INTERFACE

While the drag-and-drop ease that Appery.io's visual editor provided us when creating the user interfaces of the productivity and device-accessing applications resulted in the most productivity, the coding required for both Corona SDK and Titanium SDK for this component were trivial due to the clear documentation and matured APIs. For example, we used Corona's Widgets API to create buttons imitating the product of platform-specific code, and we also used it to have images represent the buttons. For Titanium, adding a swipe-left-to-delete handler for list rows (which is commonly used in iOS applications) was simple and only required a conditional to support the add data functionality when installed in the Android platform.

The construction of UI objects for the graphics and physics-focused game application was similar with each tool. HTML5 canvas feature was used for the PhoneGap application but it did not have as many features as the other tools. The physics aspects of the games are what work quite differently, and that will be discussed next.

We rank each tool in terms of developer user interface productivity as follows:

USER INTERFACE RANKING			
	Corona	**Titanium**	**PhoneGap**
Productivity	2	2	1
Game	1	3	2
Device-Accessing	2	2	1

Table 1. User Interface Ranking

For both our productivity and device-accessing applications, PhoneGap (via Appery.io) proved to be the tool that we found the most ease from when developing the user interface. Its visual editor was intuitive and worked for us as intended. It, though, does not offer as much of the code-level UI construction capabilities as either Corona or Titanium.

When constructing the user interface for the game application, Corona’s graphics focus allowed us to the best resources (including APIs and documentation) to include both the shape object and the floor image on screen. Despite the limitations imposed by using HTML5 canvas with

PhoneGap, Titanium’s poor graphics documentation and confusing application of Platino resulted in us still ranking PhoneGap ahead of Titanium in this area.

B. PROGRAM CONTROL

Using PhoneGap Build to wrap an HTML5 application was a simple process that only required uploading the HTML5 files to the PhoneGap server, where it would be returned as a platform-specific application build. Despite the simplicity of our game application, many lines of mathematics driven code had to be written to simulate physics, whereas APIs in Corona and Titanium did away with the extra work. We found HTML5 to be too new and we only hope that its canvas will mature into a more graphics and physics friendly development tool so that it will be capable of being used for more intensive games and simulations.

We conclude that Corona ranks ahead of Titanium in the game area mainly because of the tools’ focus towards two-dimensional graphics and physics. Appcelerator had only recently adopted and begun funding Lanica’s Platino, and the extra code required, especially that code written for tying Chipmunk 2D physics with on-screen objects, helped lean us towards Corona for gaming early in the development process.

Each of the three tools had the necessary APIs to make access to device-specific geolocation and camera features straightforward. Any JavaScript modules, REST services, or platform-specific permission changes were clearly defined in the each tools’ documentation.

Our tools' constructions of event-handling functions were similar when used in our productivity and device-accessing applications. Titanium was given the highest ranking when used for the productivity application because of the way conditionals could be used to support event-handlers when the tool is used to support a platform-specific look-and-feel, such as the different platform-specific back buttons.

Appery.io (using PhoneGap), with its very different visual development approach, enabled a high level of productivity when developing our productivity and device-accessing applications. We did not have to be concerned with syntactical errors, and although little code was necessary, there was always the option of reviewing and editing the resulting code.

Table 2 graphically represents our ranking of tools' developer program control productivity:

PROGRAM CONTROL RANKING			
	Corona	**Titanium**	**PhoneGap**
Productivity	3	2	1
Game	1	2	3
Device-Accessing	2	2	1

Table 2. Program Control Ranking

C. DATA MANAGEMENT

For the data management component, the only application where a database was used was the productivity application. Both Corona and Titanium approached this by using SQLite3 within the code itself. PhoneGap (with

Appery.io), on the other hand, required that we create a database first. Manipulating data was made convenient with REST APIs, but there was still a learning curve to applying these APIs, and local database storage is possible but requires coded JavaScript rather than just the visual editor. We found it easier to sacrifice the ease of the visual editor for use of SQLite3. We visualize our tool rankings for developer data management productivity in Table 3.

DATA MANAGEMENT RANKING			
	Corona	**Titanium**	**PhoneGap**
Productivity	1	1	2
Game	n/a	n/a	n/a
Device-Accessing	n/a	n/a	n/a

Table 3. Data Management Ranking

D. OTHER FINDINGS

Throughout the course of this thesis, we have made observations that do not quite fit into our three main metrics but, none-the-less, are still important in regards to cross-platform application developers' productivity.

Each of these three tools follows a layered architectural style. Each has developers operating at the top level by abstracting out the highest level that devices use for developer's code. This leads to savings in time, effort, and resources for companies that want their product on multiple platforms. Unfortunately, this often leads to holes in the connectors between the levels which appear when there are updates to platform layers which the cross-platform tool was not prepared to immediately adapt to. This can leave developers to a temporary drop in productivity while waiting for these updates to be supported by their tool, as in our case when we were waiting for Corona to support Android smartphones' cameras. A similar related instance was when our Xcode was altered enough during an update to prevent Titanium from building iOS applications. Our solution to this was to find and reinstall the previous Xcode update.

Device simulators were very helpful in our development of applications. The Corona simulator allowed for quick prototyping whenever we made any change, saving us from a lengthy wait for the loading of Xcode simulator or Android emulator and encouraging us to test our code often. The Corona simulator did have limitations including lack of simulating map-using applications. Titanium did not have its own simulator so if it were not for the debugging capabilities of its IDE, testing would not be as productive. PhoneGap Build and Appery.io's use of QR readers to quickly test applications on devices was very useful for ensuring that the PhoneGap "bridge" would allow applications access to device features.

VI. SUMMARY AND CONCLUSION

Whether using a commercial specialized SDK such as Corona, getting the most native look-and-feel through conditional branching with Appcelerator’s Titanium, or taking the hybrid approach with PhoneGap, each of these tools give both unique and similar productivity strengths to developers and businesses. There is no single “best” way towards cost effectiveness–the approach is different for everyone. It depends on many factors, including what a company needs now and wants later, what development requirements they have, and what the technical background of their developers are. If a business has web application developers, the best approach for them may be to use PhoneGap-related tool, taking advantage of their expertise and wrapping their product with a native wrapper. If a developer is game-focused but also wants to make business and other productivity applications, Corona-type tools may be the safest of the three approaches. Titanium is the most “native” as it uses the most native code than either of the other two, so that is an important trait to consider. If a business wants to expand their market beyond iOS and Android, then today a tool that utilizes the PhoneGap bridge—such as Appery.io—may be their best option.

By dissecting these three tools that make up significant pieces of the ever-changing mobile cross-platform development world today, we conclude that there are cost-effective development techniques that can be found in each. Thus, although there is no “one-size-fits-all” approach, a lesson for those who create cross-platform development tools would be that a productive environment should utilize many already implemented techniques, including:

- Minimizing the amount of work necessary for developers outside the realm of the development environment.
- Limiting the learning curve for those not yet oriented to the process through extensive resources, documentation, and adaption of coding techniques that developers should already be familiar with, while at the same time, not slowing them with an avalanche of unnecessary bloat.
- Ensure that key APIs and functionalities are available today and plan for the seamless addition of future APIs and functionalities as market leaders and technologies change.
- Emphasizing the importance of each stage of the cross-platform application development lifecycle when looking at overall productivity and cost-efficiency.

The DoD should consider many of these techniques while it still intends to increase the number of people able to rapidly share information through mobile technologies.

A. CLOSING COMMENTS

Mobile cross-platform application development is quite new, but already very important, and is growing and changing rapidly. Mobile cross-platform tools have been rapidly appearing in the last few years, and so there are still few books and learning resources available for any of them. Researching the productivity that developers can have when using specific tools, like we have done, is one step towards maturing the growing field. At the same time, this thesis concludes that the field has a lot more maturing and future work ahead of it.

B. FUTURE WORK

Besides the direct consideration of other smartphones that are popular in the mobile marketplace (e.g., Windows 8 Phones and Blackberry), the following are areas where in-depth research could further benefit the evolution of mobile cross-platform development.

1. Security

Mobile smartphone security and the security of personal information are important factors in developing applications whether they are available to any user or are used solely for defense-related communications and productivity. Cross-platform development tools (especially the visual drag-and-drop development environments) tend to work at high level and that can lead to developer oversight in the lower-level details that those with malicious intent could potentially take advantage of.

2. HTML5

As HTML5 specifications continue to evolve and grow, HTML5 has the potential to change the method in which developers want to build applications for mobile devices. While PhoneGap can leverage HTML5 technologies for hybrid-applications, further research can focus instead on mobile web applications, the HTML5-compliant web browsers of smartphone devices, and what smartphone features these web browsers can access now and in the future. HTML5 is already to some extent supporting mobile offline storage so that no permanent Internet access is required [28].

3. Other Tools

There are dozens of other mobile cross-platform development tools that can be analyzed to help determine which tools provide developers efficient and effective development techniques. Towards the later part of our research, we found Appery.io to, despite its use of PhoneGap, have enough uniqueness in its approach to be considered a tool in itself worth analyzing further. An example of another tool with potential is LiveCode, which allows for the creation of native applications through a drag-and-drop visual development environment, real-time “live” testing and a natural program language—English [29].

LIST OF REFERENCES

[1] Department of Defense. "*Department of Defense Commercial Mobile Device Implementation Plan*," 15 Feb. 2013. Available: www.defense.gov/news/DoDCMDImplementationPlan.pdf

[2] K. Whinnery, "*Comparing Titanium and PhoneGap*," 12 May 2012. Available: http://kevinwhinnery.com/post/22764624253/comparing-titanium-and-phonegap

[3] Appery.io, "*Appery.io*," 13 Aug. 2013. Available: http://www.appery.io

[4] VisionMobile, "*Developer Economics 2013*" June 2013. Available: http://www.DeveloperEconomics.com

[5] C. G. Acord, C. C. Murphy, "*Cross-platform Mobile Application Development A Pattern-Based Approach*," M.S. thesis, Dept. Comp. Sci., Naval Postgraduate School, Monterey, CA, 2012.

[6] B. Elgin, "Google buys Android for its mobile arsenal," *Bloomberg Businessweek*, 16 Aug. 2005. Available: http://www.businessweek.com/stories/ 2005-08-16/google-buys-android-for-its-mobile-arsenal

[7] Google, Inc., "*Platform versions*," 13 Aug. 2013. Available: http://developer.android.com/ resources/dashboard/platform-versions.html

[8] Google, Inc., "*Platform versions*," 3 Sep. 2013. Available: http://developer.android.com/design/ patterns/pure-android.html

[9] VisionMobile "*Cross-Platform Developer Tools 2012*," Feb 2012. Available: http://www.CrossPlatformTools.com

[10] Corona SDK, "*Corona Labs*," 13 Aug. 2013. Available: http://www.coronalabs.com/products/corona-sdk/

[11] W. Luh, "*Corona SDK presentation*," 8 Apr. 2012. Available: https://www.youtube.com/ watch?v=14Uz9RW2_nU

[12] Titanium Mobile Development Environment, "*Appcelerator*," 13 Aug. 2013. Available: http://www.appcelerator.com/platform/titanium-platform/

[13] PhoneGap, "*PhoneGap*," 13 Aug. 2013. Available: http://www.phonegap.com

[14] R. Ghatol, Y. Patel. *Beginning PhoneGap: Mobile web Framework for JavaScript and HTML5*. New York: Apress, 2012.

[15] S. E. Smith, "*What is control logic?*" *WiseGeek*, 9 Sep 2013. Available: http://kevinwhinnery .com/post/22764624253/comparing-titanium-and-phonegap

[16] Corona SDK, "*SDK API reference*." 13 Aug. 2013. Available: http://docs.coronalabs.com/ api/index.html

[17] B. Burton, "*Corona app development—reading from SQLite database*," 16 Feb. 2011. Available: http://www.youtube.com/watch?v=vN-5m-23zgY

[18] Corona SDK, "*Database access using SQLite*," 3 Apr. 2012. Available: http://www.coronalabs.com/blog/ 2012/04/03/tutorial-database-access-in-corona/

[19] A. Otaku, "*Add/delete items in a list with Titanium Mobile*," 11 Apr. 2012. Available: http://blog.hugeaim.com/2012/ 04/11/adddelete-items-in-the-list-with-titanium-mobile/

[20] M. Falkland, "*Corona SDK app creation: part 3*," 3 Aug. 2011. Available: http://www.youtube.com/ watch?v=brSTfZQhLz8

[21] Infinite Skills—Video Training, "*Corona SDK tutorial | creating timers*," 28 Sep. 2012. Available: http://www.youtube.com/watch?v=olE5bDVRsfE

[22] Lanica. "*Lanica for everyone!*" 7 Sep. 2013. Available: http://lanica.co/lanica-for-everyone/

[23] M. Apperson, "*Gaming comes to Titanium, introducing the Box2d module,*" Appcelerator Developer Blog, 7 Sep. 2011. Available: http://developer.appcelerator.com/blog/2011/09/gaming-comes-to-titanium-introducing-the-box2d-module.html

[24] T. Claburn, "*Appcelerator funds startup Lanica for better mobile games,*" *Information Week*, 03 Oct. 2012. Available: http://www.informationweek.com/development/mobility/appcelerator-funds-startup-lanica-for-be/240008418

[25] Lanica. "*Lanica documentation,*" 7 Sep. 2013. Available: http://docs.lanica.co/docs/

[26] Titanium Mobile Development Environment, "*Resources.*" 13 Aug. 2013. Available: https://my.appcelerator.com/resources

[27] Appery.io, "*Docs*" [Online] 23 Aug. 2013; Available: http://docs.appery.io

[28] HTML5 Rocks, "*'Offline': What does it mean and why should I care?*" 23 Aug. 2013. Available: http://www.html5rocks.com/en/tutorials/offline/whats-offline/#toc-browser-specific-features

[29] LiveCode, "*How LiveCode works.*" 23 Aug. 2013. Available: http://livecode.com/how-it-works/

THIS PAGE INTENTIONALLY BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California